Typecheck Python Programs and Find Semantic Idioms

Senxi Li Tetsuro Yamazaki Shigeru Chiba

Researchers have been empirically studying public Python programs to understand how semantic idioms are used in practice via mining software repositories. In this paper, we propose that the number of code fragments of a semantic idiom can be derived by counting and comparing type errors reported by two different type systems. The proposed method works under a comparison scheme by using two type systems with variance, typically different type kind or typing discipline: The same program can statically typecheck or not, reporting false alarms or not, when enforced by two different type systems. The different number of reported false alarms can be viewed as the approximate account of the target semantic idiom. To verify our statement, we collected 841 Python repositories on Github and investigated the frequency of how ad-hoc polymorphism is used by using existing type systems with a designed variance. Our empirical study effectively observed the expected code fragments towards the designed type system variance. Besides, our type-based mining approach also discovered some common programming practice which can be found by traditional methods such as syntactic parsing.

1 Introduction

Researchers have been empirically studying public Python programs to understand how semantic idioms are used in practice via mining software repositories. Understanding the convention of semantic idioms can help people have a better understanding their practical usage and motivates reasonable implications to developers for better language designs [3, 7, 15, 16, 19, 32]. A semantic idiom refers to code fragments that are descriptive by a particular programming concept. For instance, one interesting idiom is polymorphism, and even what specific kind of polymorphism like ad-hoc and parametric. Class inheritance and method overriding can also be good candidates.

On the other hand, semantic analysis in general and static typing in particular have been widely explored on the Python programming language with its growing popularity. Researchers have been exploring ways of adding static types to dynamic languages [5, 13], providing static guarantees so that "well-typed programs cannot go wrong" by rejecting untypeable programs conflicting with the defined type system. While language designers and researchers extend a wide variety of type systems to many applications, its combination with repository mining has so far, as far as we know, received little attention.

In this paper, We present that the number of code fragments of a semantic idiom can be derived by counting and comparing type errors reported by two different type systems. More specifically, we use two different type systems to statically

*Typecheck Python Programs and Find Semantic Idioms

Senxi Li, Tetsuro Yamazaki, Shigeru Chiba, Graduate School of Information Science and Technology, University of Tokyo.

typecheck a same program and they deliver different number of type errors, respectively. Thus, a type system variance can be deliberately designed so that the program can typecheck under a type system with the designed variance if there exists code fragments written in a particular semantic idiom, and not vice versa. Consequently, the different number of reported errors can approximately answer the amount of the target code. We believe that such type-based approach is promising and practical to achieve repository mining goals.

In summary, this paper ends up to the following contributions:

- We propose a type-based approach that can accomplish repository mining tasks of understanding semantic idioms. The method derives the approximate amount of code written in a semantic idiom by comparing the numbers of type errors under two different, static type systems.
- We materialize the proposal with a concrete type system variance, union type such that it can uncover a specific semantic idiom, ad-hoc polymorphism.
- We empirically investigate how ad-hoc polymorphism is used using the concretized method on collected Python projects.

The remaining of this paper is organized as follows: Section 2 defines what semantic idioms we want to survey in repository mining and motivates the reader by example. Section 3 explains how our proposed approach recognizes if a semantic idiom is written in the corpus. Section 4 carries out an empirical study via the proposed method on public Python repositories. A practical research question is raised in the empirical study, and we answer the question by implementing the method we propose in Section 3 with a designed, concrete type system variance. Section 5 relates our work to proceeding researches and a brief conclusion ends this paper.

2 Semantic Idioms in Python

A semantic idiom is a code fragment that serves a single semantic purpose. Finding semantic idioms in dynamic languages like Python is challenging. Yet, understanding the usage of those idioms in production can help developers comprehend the culture of the language and benefit further language designs, which is rather an attracting task in repository mining.

Here we give an example program which outlines a typical usage of ad-hoc polymorphism [17, 26], shown in List 1.

```
class Person:
  def __init__(self, name):
    self.name = name

class Vehicle:
  def __init__(self, name):
    self.name = name

def getName(x):
  return x.name

if input() == 'Person':
  x = Person('Alice')
else:
  x = Vehicle('Truck')
getName(x)
```

Listing 1 Motivating example

The program first defines two classes `Person` and `Vehicle`. A constructor method is defined by setting an instance attribute both called `name` in the method body in each of the class definitions, respectively. It then defines a function `getName` which takes one single argument. The body of the function searches for an attribute called `name`, and treats this attribute as its return. In the main part of the program, a variable `x` is initialized as either an instance of class `Person` or class `Vehicle` in a `if` statement, controlled by a runtime value read from the user keyboard. Finally, function `getName` defined before is called with the parameter `x`.

From the perspective of repository mining, one interesting topic is to empirically investigate the usage of ad-hoc polymorphism, which is a proper instance of the semantic idioms we care in this paper. In general, the investigation can be answered by counting the number of functions and methods which are implementations of ad-hoc polymorphism. In this example, the answer can be `1`, which points to the function that takes argument of two different data types defined at Line 9.

Uncovering such a semantic idiom in terms of repository mining seems not straightforward or widely studied. We believe that one of the reasons behind is simply people have not developed a proper methodology to fulfill the task in dynamic languages. Semantic idioms like ad-hoc polymorphism we quoted are code fragments that serve particular semantic purposes. It is inherently notable that one needs the knowledge of the semantics to understand its existence in a program. Nevertheless, semantic analysis on dynamic languages itself is rather complicated, and even worse on large-scale empirical studies.

Another aspect of the challenge stems from the dynamic nature of Python. For statically typed programs, a lightweight approach one can conceive is a regular lexical analysis used in many MSR researches. A scenario can go as one parses the source code and traverses the parsed trees to search type hints such that one can have a superficial understanding of semantics of the program. For instance, functions which arguments are generic-typed (e.g. in Java) are reasonably parametric polymorphic. Therefore, one can roughly know the amount of the semantic idiom written by simply counting the number of a specific kind of types in the program. While for untyped programs, this handy recipe may not give us a good taste of the mining purpose as there is little semantic information from the parsed syntax tree. As a result, the subsistence of ad-hoc polymorphic code snippet cannot be proficiently perceived using standard lexical analysis.

3 From a Perspective of Static Semantics

We present that the number of code fragments of a semantic idiom can be derived by counting and comparing type errors reported by two different type systems. The proposed approach uses static typing under two type systems with variance and they report type errors for untypable code, respectively. By comparing the number of reported type errors over the same program, it answers the approximate amount of code written in a semantic idiom.

We begin by discussing standard type inference which generates and solves type constraints, and reports type errors if the program does not typecheck under two different type systems. Then we discuss why the designed type systems and their typechecking results can answer if the program contains ad-hoc polymorphic code fragment we want to mine.

3.1 Standard Type Inference

This subsection defines one of the two type systems used in our approach to discover ad-hoc polymorphsic code fragments in an informal way. Also, it briefly explains how a standard type inference works to give types to an untyped program.

The core types of the type system is described as:

$$\tau \texttt{ ::= } \alpha \texttt{ | } \beta \texttt{ | [id: } \tau \texttt{] | } \tau \rightarrow \tau$$

τ ranges over types; α ranges over type variables; β ranges over builtin and user defined classes; `id` denotes method or attribute names; `[id:` τ`]` ranges over structural types; $\rightarrow$ denotes function types.

To typecheck untyped programs, one obvious solution is to reconstruct types by type inference. We

$$\texttt{t}_1 \leq \texttt{["name":\ t}_3\texttt{]} \quad (1)$$
$$\texttt{t}_3 \leq \texttt{t}_2 \quad (2)$$
$$\texttt{Person} \leq \texttt{t}_4 \quad (3)$$
$$\texttt{Vehicle} \leq \texttt{t}_4 \quad (4)$$
$$\texttt{t}_4 \leq \texttt{t}_1 \quad (5)$$

図 1 Generated type constraints from Line 9 to 16 in List 1

first assign unique type variables for each of the variables, function parameters and returns. Then a syntax directed analysis over the program is given to generate type constraints in a subtyping form. A constraint $\texttt{s} \leq \texttt{t}$ indicates that `s` must be a subtype of `t`. We also use the terminologies to describe the constraint such that `x` is a *lower bound* of `y` and `y` is a *upper bound* of `x`. After that, a series of resolution rules are applied to the generated type constraints so that constraints are propagated and checked if they admit a solution [22]. During the constraint propagation, if there is any inconsistent constraint, the program is judged untypable and a type error will be signaled. Otherwise, the program typechecks and a least restricted solution is computed from the lower and upper bounds for each of the type variables [4].

Back the code example in List 1, we give some of the type constraints generated by the typing disciplines of the type system, shown in Fig. 1. Parenthetically, function `getName` is typed as $\texttt{t}_1 \rightarrow \texttt{t}_2$; expression `x.name` has the type $\texttt{t}_3$; variable `x` has the type $\texttt{t}_4$.

To give a closer analysis, the constraint set suggests that $\texttt{t}_4$ should be a supertype of its two lower bounds `Person` and `Vehicle`, which conduces to a solution `object` (the top class in Python's class hierarchy). While at the same time, it might be noted that $\texttt{t}_4$ should also be a subtype of $\texttt{["name":\ t}_3\texttt{]}$ due to transitive closure from Constraint (1) and (5). That is, $\texttt{t}_4$, inferred as `object`, should have an attribute `name`, which is evidently invalid. In consequence, the program will be rejected and a type error is reported as "'`object`' type does not have attribute 'name'" at Line 10 in List 1.

3.2 Adding a Type System Variance

Described in the natural language, a type system variance is simply the difference between two type systems. A type system is defined by constructors operated on data types and a set of rules to classify syntactic phrases by their values. So, a type system variance mentioned in this paper is typically a single type kind or one syntax directed typing discipline.

To make the above program typecheck, we add union type as the type system variance to the type system described before. Furthermore, we will use that type system to infer types for the same program just as we did.

The type system now is defined as:

$$\tau \texttt{ ::= } \alpha \texttt{ | } \beta \texttt{ | [id:\ } \tau\texttt{] | } \tau \rightarrow \tau \texttt{ | } \tau \vee \tau$$

where $\vee$ ranges over union types. Notice that the only difference, or the variance between this type system and the previous one is the adaption of union type.

Same as the inference process illustrated before, the type of variable `x` will be inferred by merging its two lower bounds. Because the type system now allows union type which permits multiple primitive types for a shared instance, `x` will be inferred as $\texttt{Person} \vee \texttt{Vehicle}$. Since either type of the components of the union holds an `"name"` attribute, the type constraint $\texttt{t}_4 \leq \texttt{["name":\ t}_3\texttt{]}$ is now valid under the union type system. All the generated type constraints can be successfully resolved and the program typechecks.

3.3 Observation and Discussion

Here a detectable diversity can be observed: a difference between two type systems causes a different number of reported type errors. In the example, the program cannot typecheck with the first type system and a type error is reported. Oppositely, the program, specifically function `getName`, typecheck under the second type system which allows union type.

We claim that this observation confirms the existence of ad-hoc polymorphic code fragments in the program. That is, by counting the different number of reported type errors from the two type systems, one can approximately, but also effectively knows the number of code fragments written in the semantic idiom.

Our proposal works under the following assumptions: type kinds and typing disciplines typically enforce a particular usage of data types and assign types to a single kind of syntactic constructions, as well as how programmers commonly implement a sematic idiom. Therefore, a type kind or typing discipline can be deliberately coined as a variance between two type systems. Code fragments written in the semantic idiom will not typecheck under one type system, while the result becomes opposite if the type system variance is turned on. Still, the typechecking results of some uninterested code fragments may also alter because of the variance. We are aware of the fact that our method only tells the *approximate* amount of code written in the target semantic idiom. Under the insight, we believe that this proposed semantic analysis is promising to perform repository mining tasks of uncovering semantic idioms.

4 A First Attempt: How Ad-hoc Polymorphism is Used?

In this section, we carry out an empirical study on a set of collected Python programs from public open source. The methodology used in the study is a concrete implementation of our proposal that typechecks programs by type systems with a designed variance. Particularly, the study implements union type as the type system variance and gives empirical evidence of showing how ad-hoc polymorphism is used in practice.

In the following of this section, we provide an overview of the dataset. Moreover, a comprehensive report of the study results is presented to answer the raised research question.

4.1 Dataset Overview

To build our dataset, we first collected 841 Python repositories on Github, which contain 18,433 Python files and 1,587,530 lines of code. Those repositories were the top-starred Python repositories from the response of the Github Search API. Concurrently, because of the computing limitation of the type inferencer we implemented, we narrowed the searching results by using the search qualifier `size` so that the sizes of all resulting repositories are smaller than 5,000 KB.

We then extracted syntactically valid `.py` files from each of the collected repositories. Since our implementation only supports files written in Python 3, we further discarded Python 2 files, which resulted in 16,324 files and 1,260,287 LOC. These files are the exact input to further analysis.

4.2 How Ad-hoc Polymorphism is Used?

By comparing the typechecking results under the two designed type systems, we figured out that there were 11,100 reported type errors in difference. Since we assume that all the collected public programs do not contain any runtime error, we treat all detected type errors from our system as false positives.

To provide more details, there were 318,241 number of reported false positives by the designed type system without the variance. On the other hand,

307,141 errors were reported by the union type system. To defense the effectiveness of our designed type systems, 171,287 of 318,241, nearly 53.8%, of all the errors were path resolution failures, which means our tool failed to find the path for a module or a missing of third party library. In other word, over half of the false positives is not limitation of the type system, but rather a matter of implementation. Although different dataset were involved over studies, the ratio of non-import related errors from our implementation was quite close to that of Rak-amnouykit's investigation [23] over main stream Python static checkers Mypy [30] and Pytype [10], which was 54.2% (22,556 of 41,607) according to their qualitative report.

To understand the usage frequency of ad-hoc polymorphism in the dataset, we further enumerated the numbers of total expressions in all the input `.py` files, which was 5,058,867. By dividing the different number of false positives, the result shows that nearly 0.22% of the expressions were successfully union-typed and used in an ad-hoc polymorphic manner. A quick interpretation goes as there is one ad-hoc polymorphic usage for every 450 lines of code.

We estimate that this result is quite reasonable to public Python programs. To give a brief discussion, we postulate the fact that real-world Python programs are semantically easy in terms of data types, considering its major applications such as machine learning. It can be easily imaged that functions and methods in those programs typically operate on particular and single data types. Our results matches this hypothesis: We found that most of the functions and methods are monomorphic so that they take arguments of single data types based on our type inference result.

4.3 A Closer Look at Sampled Projects

To better investigate whether the designed type system variance can successfully discover ad-hoc polymorphic code, we provide a qualitatively closer look of what specific code fragments are located at some of the tested data.

We sampled some of the repositories in the dataset, which were obtained by precisely inspecting repositories with the most different reported type errors by the two designed type systems with the variance. In other words, the following sampled repositories are those with the most number of ad-hoc polymorphic code snippets suggested by our analysis.

Late Bindings. In *convnet-benchmarks* †1, the union type system typechecks the following piece of code:

```
if args.arch == 'alexnet':
    import alex
    model = alex.Alex()
elif args.arch == 'googlenet':
    import googlenet
    model = googlenet.GoogLeNet()
elif args.arch == 'vgga':
    import vgga
    model = vgga.vgga()
elif args.arch == 'overfeat':
    import overfeat
    model = overfeat.overfeat()
else:
    raise ValueError('Invalid
    architecture name')
# ...
model.forward()
```

In the program, variable `model` is assigned with instances of different classes in the `if` statement controlled by an input argument. `Alex`, `GoogLeNet`, `vgga` and `overfeat` are user defined classes in the corresponding modules, all of which are derived from class `Chainer` in the deep learning framework `chainer` †2. Each of the classes defines a method `forward` and the method is invoked after `model` is

†1 https://github.com/soumith/convnet-benchmarks
†2 https://chainer.org/

assigned. In our applied type system without union type, the type of variable `model` will be inferred as a union of the four types. The message passed to the variable at Line 16 can be successfully resolved since each of the type component has the requested property and thus the program typechecks. On the contrary, variable `model` is inferred as type `Chainer` under the type system without the union type variance. This time a type error is reported when the message `forward` is requested since `Chainer` class does not defined the method.

4.4 What Are The Other False Positives?

We believe that false positives that cannot typecheck by the union type system can fairly help discover other semantic idioms in dynamic languages. These false positives are assumed not meaningless but rather of valuable indications which can serve for mining purposes of discovering Pythonic code fragments. As a further analysis, we substantiated our claim by sampling and show evidence by concrete examples.

We randomly sampled 58 repositories which contained 181,444 LOC in the dataset and manually inspected the false positives reported by the union type system. With further analysis, 5,319 of the errors were errors that failure of resolution of a union type. We took a closer look at the these places and inspected the untyped code fragments to figure out why a union type cannot be successfully given to the expressions, and to verify if these code fragments match our expectation. We list them in categories below.

Mixed types in containers. This coding exercise describes elements of different data types in container objects such as list and dictionary. As shown in 2 discovered in repository *datasets* †3,

```
1 cfg = {'rsync_cmd': 'rsync', '
      rsync_sync_opts': '-avz', '
      rsync_list_opts': '--list-only', '
      data_store': 'fpbatta@tompouce.
      science.ru.nl', 'subdirs_as_datasets
      ': False}
```

Listing 2 Code in sampled repository

the type of the key of the dictionary expression will be inferred as `String` under the designed union type system. Besides, the type of its value will be infered as `String` ∨ `Bool`, which is the union of its all elements. Thus, if one of the values is retrieved by from this dictionary object, a type error will be raised when a massage not shared by the multiple types is passed. Namely,

```
1 self.source_location = self.config['
      data_store'] + ':' + self.source_dir
```

where the value of `self.config` is equal to the value of `cfg` in List 2. Our type system blamed that "bad operand for `+`: `Int` and `String`", while this is an expected behavior of the designed, static type system though. Similar code snippets that could not typecheck with our union type system are also found in *QuakeHordes* †4 and *HelloGitHub* †5.

Reassigning variables with different types. In repository *BeautifulDiscord* †6, we found another common coding pattern that cannot typecheck by the union type system.

```
1 index = input("Discord executable to use
      (number): ")
2 try:
3     index = int(index)
4 except ValueError as e:
5     print('Invalid index passed')
```

Variable `index` is first assigned by getting input from the user. After that, the same variable is reassigned to `Int` at Line 3 above. Therefore, the type system will complain the usage of variable `index` to the place where an `Int` is expected in the program

†3 https://github.com/fpbattaglia/datasets

†4 https://github.com/mUogoro/QuakeHordes

†5 https://github.com/521xueweihan/HelloGitHub

†6 https://github.com/leovoel/BeautifulDiscord

later.

5 Related Works

In this section, we will review some of the existing studies and make a brief discussion comparing with our proposal, respectively.

Monat et al. [18] derived a sound, static analysis by abstract interpretation by giving Python programs a concrete semantics, and further taking full control-flow into account such as supporting exception handling and expressing parametric polymorphism. Compared with their work of performing static analysis by structural induction on the syntax, our proposal applies a standard static type checking, which is generally flow-insensitive and context-insensitive, with a constraint-based type inference for dynamically typed programs. Furthermore, our approach performs repository mining tasks by carefully designing several, different type systems and uncovering a semantic idiom with the corresponding type system variance.

Rak-amnouykit et al. [23] did an empirical sytudy of how Python developers use type annotations with PEP484 [30], and evaluated the performance of Mypy [14] and Pytype [10] on public Github repositories. Mypy and Pytype are two of the celebrated tools of performing static type checking and inference for Python programs. The authors explored that programs with type annotations rarely typecheck by the existing tools: both of them exhibit false positives and also useful errors. From the aspect of empirically studying open source Python repositories, our work also enforces type systems to statically typecheck Python programs. Conversely, we offer our own implementation so that we can easily apply and manage different type systems to discover semantic idioms. Besides, reported false positives are viewed as indication of the target code fragments suggested by the designed variance of two type systems.

6 Conclusion

In this paper we proposed a type-based approach which can accomplish repository mining tasks of uncovering semantic idioms. The proposed approach works by using two different type systems with designed variance so that program typechecks or not while switching the variance. By comparing the number of reported false alarms, one can derive the approximate amount of code that is written in the corresponding semantic idiom.

We carried out an empirical study over 841 public Python repositories to investigate common programming idioms. Our approach successfully detected the usage frequency of ad-hoc polymorphism, but also it uncovered some well-known coding practice that can be found by traditional methods like syntactic parsing. Thus, we believe that such typed-based method is an effective and promising approach to performing repository mining tasks.

参 考 文 献

[1] Akbar, S. A. and Kak, A. C.: A Large-Scale Comparative Evaluation of IR-Based Tools for Bug Localization, *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, New York, NY, USA, Association for Computing Machinery, 2020, pp. 21–31.

[2] Allamanis, M. and Sutton, C.: Mining Idioms from Source Code, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, New York, NY, USA, Association for Computing Machinery, 2014, pp. 472–483.

[3] Amanatidis, T. and Chatzigeorgiou, A.: Studying the evolution of PHP web applications, *Information and Software Technology*, Vol. 72(2016), pp. 48–67.

[4] An, J.-h. D., Chaudhuri, A., Foster, J. S., and Hicks, M.: Dynamic Inference of Static Types for Ruby, *SIGPLAN Not.*, Vol. 46, No. 1(2011), pp. 459–472.

[5] Aycock, J.: Aggressive Type Inference, November 1999.

[6] Cousot, P.: Types as Abstract Interpretations, *Proceedings of the 24th ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, POPL '97, New York, NY, USA, Association for Computing Machinery, 1997, pp. 316–331.

[7] Dyer, R., Rajan, H., Nguyen, H. A., and Nguyen, T. N.: Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features, *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, New York, NY, USA, Association for Computing Machinery, 2014, pp. 779–790.

[8] Garcia, R., Clark, A. M., and Tanter, E.: Abstracting Gradual Typing, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, New York, NY, USA, Association for Computing Machinery, 2016, pp. 429–442.

[9] github: octoverse, May 2020.

[10] google: pytype, May 2016.

[11] Guido van Rossum, Barry Warsaw, N. C.: PEP 8 – Style Guide for Python Code, July 2001.

[12] Kagdi, H., Collard, M. L., and Maletic, J. I.: A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution, *J. Softw. Maint. Evol.*, Vol. 19, No. 2(2007), pp. 77–131.

[13] Kazerounian, M., Ren, B. M., and Foster, J. S.: Sound, Heuristic Type Annotation Inference for Ruby, *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2020, New York, NY, USA, Association for Computing Machinery, 2020, pp. 112–125.

[14] Lehtosalo, J., van Rossum, G., and Levkivskyi, I.: mypy, June 2012.

[15] Marcilio, D. and Furia, C. A.: How Java Programmers Test Exceptional Behavior, *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 207–218.

[16] Mazinanian, D., Ketkar, A., Tsantalis, N., and Dig, D.: Understanding the Use of Lambda Expressions in Java, *Proc. ACM Program. Lang.*, Vol. 1, No. OOPSLA(2017).

[17] Milner, R.: A theory of type polymorphism in programming, *Journal of Computer and System Sciences*, Vol. 17, No. 3(1978), pp. 348–375.

[18] Monat, R., Ouadjaout, A., and Miné, A.: Static Type Analysis by Abstract Interpretation of Python Programs, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, Hirschfeld, R. and Pape, T.(eds.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166, Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 17:1–17:29.

[19] Nakamaru, T., Matsunaga, T., Yamazaki, T., Akiyama, S., and Chiba, S.: An Empirical Study of Method Chaining in Java, *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, New York, NY, USA, Association for Computing Machinery, 2020, pp. 93–102.

[20] Oracle: JDK 5.0 Documentation, September 2004.

[21] Oracle: Java Platform, Standard Edition 8 API Specification, March 2014.

[22] Pottier, F.: A Framework for Type Inference with Subtyping, *SIGPLAN Not.*, Vol. 34, No. 1(1998), pp. 228–238.

[23] Rak-amnouykit, I., McCrevan, D., Milanova, A., Hirzel, M., and Dolby, J.: Python 3 Types in the Wild: A Tale of Two Type Systems, *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2020, New York, NY, USA, Association for Computing Machinery, 2020, pp. 57–70.

[24] Siek, J. and Taha, W.: Gradual Typing for Objects, *ECOOP 2007 – Object-Oriented Programming*, Ernst, E.(ed.), Berlin, Heidelberg, Springer Berlin Heidelberg, 2007, pp. 2–27.

[25] Siek, J. G. and Taha, W.: Gradual Typing for Functional Languages, *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, 2006, pp. 81–92.

[26] Strachey, C.: Fundamental Concepts in Programming Languages, August 1967.

[27] Thatte, S.: Quasi-Static Typing, *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, New York, NY, USA, Association for Computing Machinery, 1989, pp. 367–381.

[28] Tobin-Hochstadt, S. and Felleisen, M.: Interlanguage Migration: From Scripts to Programs, *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, New York, NY, USA, Association for Computing Machinery, 2006, pp. 964–974.

[29] Turnu, I., Concas, G., Marchesi, M., Pinna, S., and Tonelli, R.: A modified Yule process to model the evolution of some object-oriented system properties, *Information Sciences*, Vol. 181, No. 4(2011), pp. 883–902.

[30] van Rossum, G., Lehtosalo, J., and Langa, L.: PEP 484 – Type Hints, September 2014.

[31] Zapponi, C.: GitHut, May 2014.

[32] Zhai, H., Casalnuovo, C., and Devanbu, P. T.: Test Coverage in Python Programs, *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, (2019), pp. 116–120.