Python Probabilistic Type Inference with Natural Language Support

Zhaogui Xu*, Xiangyu Zhang†‡, Lin Chen*, Kexin Pei†, and Baowen Xu*‡

* State Key Laboratory of Novel Software Technology
* Nanjing University, China

† Department of Computer Science
† Purdue University, USA

zgxu@smail.nju.edu.cn {xyzhang, kpei}@cs.purdue.edu {lchen, bwxu}@nju.edu.cn

ABSTRACT

We propose a novel type inference technique for Python programs. Type inference is difficult for Python programs due to their heavy dependence on external APIs and the dynamic language features. We observe that Python source code often contains a lot of type hints such as attribute accesses and variable names. However, such type hints are not reliable. We hence propose to use probabilistic inference to allow the beliefs of individual type hints to be propagated, aggregated, and eventually converge on probabilities of variable types. Our results show that our technique substantially outperforms a state-of-the-art Python type inference engine based on abstract interpretation.

CCS Concepts

•Software and its engineering → Automated static analysis; •Mathematics of computing → Max marginal computation;

Keywords

Python; Dynamic Languages; Type Inference; Probabilistic Inference

1. INTRODUCTION

Python is one of the most popular programming languages nowadays. However, since Python is a dynamic language, developers often suffer from the lack of type information during development. In fact, type errors are very commonly encountered bugs in Python. However, Python type inference is highly challenging. Many Python projects heavily utilize external API functions that are often not in Python. Objects are created and mutated in those API functions. They may also become correlated by these functions (e.g., through aliasing). Such creations, mutations, and correlations are critical for correct type inference. However, these API functions are usually difficult to analyze due to the different programming languages used and the lack of source code. The large number of API functions also makes manual mocking prohibitively expensive. In addition, Python variable types are path-sensitive. A variable may have various types depending on the program paths. Types and attribute sets of objects can be dynamically mutated, substantially adding to the difficulty of static typing.

Traditional unification based type inference is not applicable to Python type inference due to path-sensitivity. Researchers have proposed various solutions for Python type inference [7, 8, 5, 9, 6]. Most of them work by leveraging data flow between untyped variables and variables of known types (e.g., variables assigned with constants). Their effectiveness hinges on the manual mocking of the large number of external API functions. According to our experiment (Section 6), a state-of-the-art system *PySonar2* [1] that was used by Google Inc. can only type 49.47% of variables in real-world programs. There have been a lot of works on type inference for dynamic languages in general [26, 28, 23, 24]. Many of them have the similar idea of leveraging variables with known types. Some of them are dynamic analysis, requiring good test coverage.

We propose a novel Python type inference technique based on probabilistic inference. We observe that Python programs have a lot of type hints such as attributes that are accessed, variable names, and explicit type checks. However, many of these type hints are uncertain, meaning that they are not fully reliable. For example, attribute sets may be dynamically mutated so that the observed attribute accesses may not match the prototype of the type. Developers may not respect the naming conventions. Our idea is to correlate all these uncertain type hints through probabilistic inference, which is a technique that allows beliefs, that is, the initial confidences of type hints, to be propagated and aggregated through the correlations among program artifacts (e.g., data flow between variables). Eventually, the computation converges on the probabilities of variable types.

Our contributions are summarized as follows.

- We identify four kinds of type hints that can be used to infer types in Python, including data flow between variables, attribute accesses, variable names, and explicit type checks. Some of these hints are uncertain.
- We propose the novel idea of using probabilistic inference for Python type inference, which allows us to naturally model the uncertainty of type hints and easily handle dynamic features.
- We develop a machine learning based approach to ex-

‡ Corresponding authors.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950343

```
1: def gzip(f, *args, **kwargs):
2:   resp = f(*args, **kwargs)
3:   url = resp.url
4:   mthd = resp.method
5:   data = compress(resp)
     ...
6:   result = resp
7:   return result
```

Figure 1: A motivating example.

```
1: def deflate(f, *args, **kwargs):
2:   r = f(*args, **kwargs)
3:   if not args:
4:     data = r.data
5:   else:
6:     req = r.request
     ...
7:   return r
```

Figure 2: An example for path sensitivity.

tract variable type hints from their names.

- We develop a prototype and evaluate it on 18 real-world Python projects. Our results show that our technique can type 79.09% of the variables that cannot be typed by a state-of-the-art system with estimated 82.86% precision.

2. CHALLENGES

Python program type inference has the following prominent challenges.

Incomplete Data Flow and Lack of Interface Definition. Most existing static type inferencing techniques work by observing data flow starting from values with known types [1, 7, 23, 28]. Unfortunately, Python programs heavily rely on external functions that may be implemented in other languages and have undocumented interfaces/side-effects. Moreover, many Python projects are indeed libraries that provide services to other downstream projects that may not be available during type analysis. All these lead to incomplete data flow and difficulties in type inference.

Consider a Python program snippet shown in Fig. 1. It is extracted and adopted from a popular Python library *httpbin*. As we can see from the example, function `f()` is a parameter of function `gzip()`. In the library, `gzip()` is a top level function that is not invoked by any other functions. As such, we cannot get any type information for the parameters of `gzip()` from function invocations. Hence, we do not know the interface of `f()`, including the type of return value. In addition, an external function `compress()` is invoked at line 5, which does not have any Python source code. Therefore, it is difficult to know if the external function has any side-effects on the object referenced by `resp`. Most existing techniques will fail to infer the types of `resp`, `data` and `result` due to the incompleteness of data flow.

The overarching idea of our technique is to *leverage the incomplete/uncertain type hints in a program and conduct probabilistic inference*. For each variable, our tool will produce a list of types that is ordered by their likelihood. In particular, we leverage two kinds of type hints. We call them *hints* because they are uncertain/incomplete by nature. The first kind is extracted from program semantics, describing how a variable is being used and what attributes are being accessed. Note that such accesses are incomplete in most cases (i.e., only a subset of attributes of an object are accessed). The second kind is variable names. We assume the developers for a project follow some naming conventions, which can be extracted by data mining the names of variables that can be typed by existing type inference tools. Note that these naming hints are uncertain. We can only say "*this variable may likely be X type according to its name*". In our example, according to lines 3 and 4, we know that the object referenced by `resp` must contain attributes "`url`" and "`method`". However, from these two hints, we cannot determine the type of `resp` because both the instances of type `Response` and the instances of type `Request` have these two attributes. In addition, we do not know if other attributes of `resp` are accessed in the function `compress ()`. Interestingly, from the naming convention, variable `resp` may possibly be an instance of `Response`, as it is lexically similar to "Response". In other words, when we consider both kinds of hints, the likelihood of `resp` being of `Response` type becomes much higher.

Path Sensitivity and Dynamic Updates. As Python is a dynamic language, a variable may have different types at runtime, depending on the program paths. These types may not even have sub-type relations. In such cases, static type inference should not produce a specific type for x at location l, but rather a type set. Path sensitivity makes probabilistic inference particularly challenging as contradicting type hints may be collected along different paths. If their probabilities are simply aggregated, they may nullify each other, leading to type inference failures or incorrect types. Consider an example in Fig. 2. Variable r has different types along the two branches. We observe two attribute accesses of `r`, i.e., accessing `data` and `request` along the two respective branches. We cannot simply assert that the type of `r` contain both attributes. Otherwise, the probabilistic inference engine would fail to infer the type of `r` as there does not exist any type (in the type domain) that have both attributes.

The set of attributes of an object can also be dynamically modified at runtime so that the type hints collected along a path may not match with the original prototype of an object. Probabilistic inference needs to take this into consideration.

We discuss the probabilistic inference engine in Section 4. We then further discuss how to handle path sensitivity and dynamic updates in Section 5.2.

3. SYSTEM OVERVIEW

In this section, we give an overview of our system.

Framework. Fig. 3 presents the architecture of our system. It consists of three components: the variable name classifier, the probabilistic constraint generator and the probabilistic type inference engine. Basically, our technique takes the whole project source code, and then outputs a ranked list of types for each variable, each type annotated with its probability. The high level work-flow of our approach contains the following steps. First, it extracts the naming conventions of the project by performing machine-learning on the variables that can be typed with existing static type inference techniques. Second, it generates a set of constraints from the data flow, attribute accesses, type check predicates, and variable names. Third, it transforms these generated constraints to a probabilistic inference network, called *factor graph*. Fourth, it resolves the graph using *belief propagation* to get the probabilities of individual types for each variable. Finally, it ranks the computed type probabilities (for each variable) and filters out the unlikely types according to some given thresholds.

Variable Name Classifier. The classifier takes the source code of a Python project and applies an existing static type

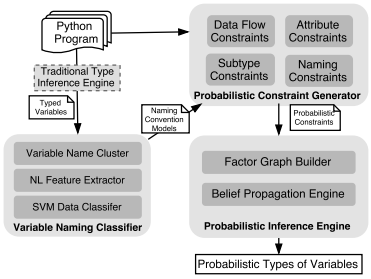

Figure 3: System framework.

inference technique [1, 7, 23, 28]. The variables that can be typed are fed to a machine learning engine that takes both the variable names and the corresponding types, and outputs a trained type classification model based on the naming conventions of the project.

Probabilistic Constraint Generator. We generate four kinds of probabilistic constraints, namely, *data flow constraints*, *attribute constraints*, *subtype constraints* and *naming constraints*. Let us consider the motivating example. Given the definitions presented in Fig. 4, assume our goal is to probabilistically infer the types of variable `result`. Our approach computes the probability of `result` having each type in the type domain $TypeDomain$. As part of the procedure, given type $Response \in TypeDomain$, our technique aims to infer the probability of `result` being a variable of the `Response` type. Fig. 5 shows the corresponding constraints.

First, it generates constraints from program data flow. According to line 6, we can generate an equivalence constraint (a) between variables `resp` and `result` with 1.0 probability. Intuitively, it means the types of `resp` and `result` *must* be equivalent, according to the data flow.

Second, we generate probabilistic constraints from the attribute accesses, called *attribute constraints*. The constraints shown in (b) represent the attribute constraints between `resp` and type `Response`. The first constraint in (b) denotes that if *resp* holds an instance of `Response`, the attribute set of type `Response` must contain the attributes "`url`" and "`method`" (i.e. with a probability $HIGH = 0.95$). The constraint is constructed from lines 3 and 4. It is a standard in probabilistic inference to use a close to 1.0 value to denote high probability when there is uncertainty [11]. In this case, the uncertainty comes from the dynamic attribute updates of an object. Hence, we can only say an instance of `Response` has a high probability containing attributes "`url`" and "`method`". The second constraint in (b) represents if *resp* contains the two attributes "`url`" and "`method`", we have a probability 0.8 that `resp` is an instance of `Response`. We use probability because there are other types in the domain that also have the two attributes (e.g., `Request`). Note that we do not use probability 0.5, which has the meaning of "*there is no evidence that* `resp` *is an instance of* `Response`". In contrast, a probability less than 0.5 means that there is evidence that the statement is not true.

Third, Python programs often use explicit runtime type check functions such as `isinstance()` to test the dynamic type of an object. A path sensitive analysis that can distinguish the type of an object according to the branch taken can also provide strong type hints. We model such hints as subtype constraints (See details in Section 5.3).

Fourth, we generate probabilistic constraints according to the variable naming conventions, called *naming constraints*. We assume that each variable in the program follows certain

$x \in VariableSet \qquad \tau \in TypeDomain$
$\mathcal{A}(\tau)$ represents the attribute set of the original prototype of τ.
$\mathcal{P}(x, \tau)$ represents a predicate denoting the type of variable x can be type τ.
$\mathcal{N}(x, \tau)$ represents a predicate denoting the type of variable x can be type τ from the naming conventions of type τ.
$\mathcal{C} : a \xrightarrow{p} b$ represents a probabilistic constraint denoting predicate a has a probability p implying predicate b.

Figure 4: Basic definitions.

(a)	$\mathcal{P}(resp, Response) \xleftrightarrow{1.0} \mathcal{P}(result, Response)$
(b)	$\mathcal{P}(resp, Response) \xrightarrow{0.95} (\{\text{“}url\text{”}, \text{“}method\text{”}\} \subset \mathcal{A}(Response)$ $(\{\text{“}url\text{”}, \text{“}method\text{”}\} \subset \mathcal{A}(Response)) \xrightarrow{0.8} \mathcal{P}(resp, Response)$
(c)	$\mathcal{N}(resp, Response) \xrightarrow{0.7} \mathcal{P}(resp, Response)$ $\mathcal{N}(result, Response) \xrightarrow{0.7} \mathcal{P}(result, Response)$

Figure 5: Generated constraints of the example.

naming conventions specific to the project. The constraints in Fig 5 (c) state if `resp`/`result` is an instance of `Response` according to the naming conventions, there is 0.7 probability that `resp`/`result` is of type `Response`. The probability is to model the uncertainty that the developers may not always obey the naming conventions.

Probabilistic Inference Engine. We conjoin the four kinds of constraints into a probabilistic graphical model, called *factor graph*, and perform *belief propagation* using the *sum-product* algorithm. Belief propagation is an iterative procedure that will eventually produce a satisfying marginal probability of $\mathcal{P}(result, Response)$, which indicates the probability of variable `result` having type `Response`. We provide the details of the probabilistic model in next section. Such probability is computed for each type τ in the type domain. For example, the probability of `result` being of string type is very low as the probabilities of constraints in Fig. 5 (b) become close to 0. Eventually, a ranked list of types is generated for each variable at each program point.

4. PROBABILISTIC INFERENCE

In this section, we illustrate how probabilistic inference is performed via the motivating example in Fig. 1. Assume our goal is to infer the probability of the type of variable `result` being `Response`. Let boolean variables x_1 and x_2 denote the predicates $\mathcal{P}(result, Response)$ and $\mathcal{P}(resp, Response)$ in Fig. 5, respectively. We then can denote the original constraint (a) as follows.

$$x_1 \xrightarrow{1.0} x_2 \wedge x_2 \xrightarrow{1.0} x_1 \tag{1}$$

Let boolean variable x_3 represent the predicate $\{\text{“}url\text{”}, \text{“}method\text{”}\} \subset \mathcal{A}(Response)$ in Fig. 5 (b), we have the following formula.

$$x_2 \xrightarrow{0.95} x_3 \wedge x_3 \xrightarrow{0.8} x_2 \tag{2}$$

With the equations (1) and (2), given the observation $x_3 = 1$, we want to infer the probability for $x_1 = 1$. Intuitively, we aim to compute the likelihood of `result` being an instance of `Response` when we observe `resp` contains both "`url`" and "`method`" attributes.

In addition, let boolean variables x_4 and x_5 represent the predicates $\mathcal{N}(resp, Response)$ and $\mathcal{N}(result, Response)$ in Fig. 5, respectively. We thus have the following formula.

$$x_4 \xrightarrow{0.7} x_2 \wedge x_5 \xrightarrow{0.7} x_1 \tag{3}$$

Formally, we represent each of the aforementioned probabilistic constraints $\mathcal{C}_i$ as a probabilistic function $\mathcal{F}_i$ as follows.

$\mathcal{C}_1 : x_1 \xrightarrow{1.0} x_2$ $\quad \mathcal{C}_2 : x_2 \xrightarrow{1.0} x_1$ $\quad \mathcal{C}_3 : x_2 \xrightarrow{0.95} x_3$

$\mathcal{C}_4 : x_3 \xrightarrow{0.8} x_2$ $\quad \mathcal{C}_5 : x_4 \xrightarrow{0.7} x_2$ $\quad \mathcal{C}_6 : x_5 \xrightarrow{0.7} x_1$

$\mathcal{C}_7 : x_3 = 1(1.0)$ $\quad \mathcal{C}_8 : x_4 = 1(0.8)$ $\quad \mathcal{C}_9 : x_5 = 1(0.4)$

$$\mathcal{F}_1(x_1,x_2) = \begin{cases} 1.0 & \text{if } (x_1 \to x_2) = 1 \\ 0.0 & \text{otherwise} \end{cases} \quad \mathcal{F}_6(x_1,x_5) = \begin{cases} 0.7 & \text{if } (x_5 \to x_1) = 1 \\ 0.3 & \text{otherwise} \end{cases}$$

$$\mathcal{F}_2(x_1,x_2) = \begin{cases} 1.0 & \text{if } (x_2 \to x_1) = 1 \\ 0.0 & \text{otherwise} \end{cases} \quad \mathcal{F}_7(x_3) = \begin{cases} 1.0 & \text{if } x_3 = 1 \\ 0.0 & \text{otherwise} \end{cases}$$

$$\mathcal{F}_3(x_2,x_3) = \begin{cases} 0.95 & \text{if } (x_2 \to x_3) = 1 \\ 0.05 & \text{otherwise} \end{cases} \quad \mathcal{F}_8(x_4) = \begin{cases} 0.8 & \text{if } x_4 = 1 \\ 0.2 & \text{otherwise} \end{cases}$$

$$\mathcal{F}_4(x_2,x_3) = \begin{cases} 0.8 & \text{if } (x_3 \to x_2) = 1 \\ 0.2 & \text{otherwise} \end{cases} \quad \mathcal{F}_9(x_5) = \begin{cases} 0.4 & \text{if } x_5 = 1 \\ 0.6 & \text{otherwise} \end{cases}$$

$$\mathcal{F}_5(x_2,x_4) = \begin{cases} 0.7 & \text{if } (x_4 \to x_2) = 1 \\ 0.3 & \text{otherwise} \end{cases}$$

Figure 6: Constraints and valuation functions.

Table 1: Boolean constraints with probabilities.

x_2	x_3	x_4	$\mathcal{F}_3(x_2,x_3)$	$\mathcal{F}_4(x_2,x_3)$	$\mathcal{F}_5(x_2,x_4)$	$\mathcal{F}_7(x_3)$	$\mathcal{F}_8(x_4)$
0	0	0	0.95	0.8	0.7	0.0	0.2
0	0	1	0.95	0.8	0.3	0.0	0.8
0	1	0	0.95	0.2	0.7	1.0	0.2
0	1	1	0.95	0.2	0.3	1.0	0.8
1	0	0	0.05	0.8	0.7	0.0	0.2
1	0	1	0.05	0.8	0.7	0.0	0.8
1	1	0	0.95	0.8	0.7	1.0	0.2
1	1	1	0.95	0.8	0.7	1.0	0.8

$$\mathcal{F}_i(x_1,...,x_m) = \begin{cases} p & \text{if the constraint is } true \\ 1-p & \text{otherwise} \end{cases} \quad (4)$$

where $x_1, x_2, ..., x_m$ represent the boolean variables associated with the constraint $\mathcal{C}_i$ and p represents the probability of the constraint being *true*.

For our motivating example, the probabilistic constraints $\mathcal{C}_i (i = 1, ..., 9)$ and the corresponding probabilistic functions $\mathcal{F}_i (i = 1, ..., 9)$ are listed in Fig. 6. Note that, according to lines 3 and 4 of the motivating example, the attribute set of *resp* must contain "`url`" and "`method`". Therefore, the probability of $x_3 = 1$ ($\mathcal{C}_7$) is 1.0. The probabilities of $\mathcal{C}_8$ and $\mathcal{C}_9$ are predicted by the variable name classifier according to the naming conventions. We will explain how they are computed in Section 5.

In general, assume there are k probabilistic constraints $\mathcal{C}_1$, $\mathcal{C}_2$,..., $\mathcal{C}_k$ on n boolean variables x_1, x_2, ..., x_n. Functions $\mathcal{F}_1$, $\mathcal{F}_2$, ..., $\mathcal{F}_k$ correspond to these constraints. Since all the constraints ought to be satisfied, the function representing the conjunction of the constraints is hence the product of the corresponding probabilistic functions. Therefore, we have the following equation.

$$\mathcal{F}(x_1, x_2, ..., x_n) = \mathcal{F}_1 \times \mathcal{F}_2 \times ... \times \mathcal{F}_k \quad (5)$$

The joint probability function is defined as follows [33], which is essentially the normalized version of $\mathcal{F}(x_1, ..., x_n)$.

$$p(x_1, x_2, ..., x_n) = \frac{\mathcal{F}_1 \times \mathcal{F}_2 \times ... \times \mathcal{F}_k}{\sum_{x_1,...,x_n} (\mathcal{F}_1 \times \mathcal{F}_2 \times ... \times \mathcal{F}_k)} \quad (6)$$

According to the above equation, we can further compute the marginal probability $p_i(x_i)$ as follows.

$$p(x_i) = \sum_{x_1} \sum_{x_2} ... \sum_{x_{i-1}} \sum_{x_{i+1}} ... \sum_{x_n} p(x_1, x_2, ..., x_n) \quad (7)$$

In other words, the marginal probability is the sum over all the variables other than x_i [33]. For a type τ_i, variable x_i asserts that the type of a given program variable be τ_i. Hence, in order to discover the possible types of the variable, we compute x_i's probability for all the τ_i's in the type domain and order them by their marginal probabilities.

Consider the motivating example. For simplicity, we ignore the program variable `result` and only consider variable `resp`. The related boolean variables are thus x_2, x_3 and x_4,

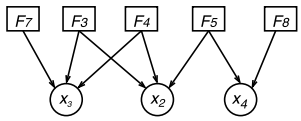

Figure 7: Factor graph example.

and the corresponding probabilistic functions are $\mathcal{F}_3$, $\mathcal{F}_4$, $\mathcal{F}_5$, $\mathcal{F}_7$ and $\mathcal{F}_8$. Table 1 presents the values of the probabilistic functions. Assume we want to compute the marginal probability $p(x_2 = 1)$, that is, the probability of the type of variable `resp` being `Response`. The computation procedure is shown by the following equation.

$$\begin{aligned} p(x_2 = 1) &= \frac{\sum_{x_3,x_4} \mathcal{F}_3(1,x_3) \times \mathcal{F}_4(1,x_3) \times \mathcal{F}_5(1,x_4) \times \mathcal{F}_7(x_3) \times \mathcal{F}_8(x_4)}{\sum_{x_2,x_3,x_4} \mathcal{F}_3(x_2,x_3) \times \mathcal{F}_4(x_2,x_3) \times \mathcal{F}_5(x_2,x_4) \times \mathcal{F}_7(x_3) \times \mathcal{F}_8(x_4)} \\ &= \frac{0.05 \times 0.8 \times 0.7 \times 0.0 \times 0.2 + ... + 0.95 \times 0.8 \times 0.7 \times 1.0 \times 0.8}{0.95 \times 0.8 \times 0.7 \times 0.0 \times 0.2 + ... + 0.95 \times 0.8 \times 0.7 \times 1.0 \times 0.8} \\ &= \frac{0.532}{0.6042} = 0.8805 \end{aligned} \quad (8)$$

Similarly, when program variable `result` is also considered, we can compute the marginal probability of $p(x_1 = 1) = 0.9052$, representing that the type of `result` has a probability 0.9052 being `Response`. On the other hand, let us consider the probability of the type of `result` being `Request`. Since the variable `resp` is less likely the `Request` type following the naming conventions, the name classifier computes a low probability 0.3 for $\mathcal{N}(resp, Request)$. With other constraints, we can hence infer the probability of `result` being a `Request` type variable $p'(x_1 = 1) = 0.8622$, which is lower than the probability of the `Response` type.

Factor Graph. The computation of marginal probabilities via Equation 7 could be very expensive as it has to enumerate all the possible combinations of the variable valuations. *Factor Graph* [33] is a probabilistic graphical model allowing efficient computation. We present a part of the factor graph of our previous example in Fig. 7. A *factor graph* consists of two kinds of nodes: *factor* nodes represented by squares and *variable* nodes represented by circles. A *factor node* represents a probabilistic function, e.g., $\mathcal{F}_i$ in Equation 7. A *variable node* represents a variable in the function, e.g., x_i in Equation 7. Edges are directed from a factor node to each variable of the function.

Sum-Product Algorithm. The *sum-product* algorithm [33] is an efficient algorithm computing marginal probabilities based on *factor graph*. In this algorithm, probabilities are propagated only between adjacent nodes through message passing. The probability of a node is updated by integrating all the messages it receives. Then the node will further propagate the probability to its receivers. The algorithm is iterative and terminates when the probabilities converge. Probabilistic inference has been successfully applied to many areas such as debugging and artificial intelligence [15, 11, 16, 12]. In this paper, our prototype tool is built upon *pgmpy* [2], an open source probabilistic graphical modeling library in Python.

5. CONSTRAINT GENERATION

In this section, we discuss how to generate the probabilistic constraints used by the type inference engine. As discussed in Section 3, the probabilistic constraints fall into four categories: (1) *data flow constraints* generated from data flow of the program; (2) *attribute constraints* extracted according to attribute accesses; (3) *subtype constraints* that

Code	Constraint
1: y^1 = x^1	$\mathcal{P}(x^1,\tau) \overset{1.0}{\longleftrightarrow} \mathcal{P}(y^1,\tau)$
2: **if** a^1 > 0:	
3: z^1 = y^1	$\mathcal{P}(y^1,\tau) \overset{1.0}{\longleftrightarrow} \mathcal{P}(z^1,\tau)$
5: **else**:	
6: z^2 = b^1	$\mathcal{P}(b^1,\tau) \overset{1.0}{\longleftrightarrow} \mathcal{P}(z^2,\tau)$
7: z^3 = $\phi(z^1, z^2)$	$\mathcal{P}(z^3,\tau) \overset{1.0}{\longleftrightarrow} \mathcal{P}(z^1,\tau) \vee \mathcal{P}(z^2,\tau)$
8: w^1 = z^3	$\mathcal{P}(z^3,\tau) \overset{1.0}{\longleftrightarrow} \mathcal{P}(w^1,\tau)$
9: s^1 = w^1	$\mathcal{P}(s^1,\tau) \overset{1.0}{\longleftrightarrow} \mathcal{P}(w^1,\tau)$

Figure 8: An illustrative example.

denote the explicit runtime type checking conditions; (4) *naming constraints* derived from variable names and the naming conventions of the project. Note that all these constraints are transformed into *factor graphs*, from which the type probabilities are computed for each variable.

5.1 Data Flow Constraints

Data flow constraints denote the define-use relations across variables. It is particularly useful for inferring types of a variable when its attributes are never or rarely accessed but it has def-use relations with other variables whose attribute accesses provide lots of type hints. In many cases, data flow constraints also allow our engine to aggregate the attribute access type hints across multiple correlated variables. In our motivating example Fig. 1, there is no direct evidence that indicates the type(s) of the variable `result`. However, we can still compute its type(s) due to its data flow correlation with variable `resp`.

We analyze each module of the given Python project and build the module dependence graph. According to the module dependence graph, we compute a topological order for all the involved modules. For each module, we first transform the source code into the *Single Static Assignment* (SSA) form, in which each variable is defined exactly once. We leverage ϕ functions to merge analysis results from different branches. For each assignment $y = op(x_1, x_2, ...)$ in the program, we generate a data flow constraint as follows.

$$\mathcal{P}(y,\tau) \overset{1.0}{\longleftrightarrow} \bigvee_{x_i} \mathcal{P}(x_i,\tau) \tag{9}$$

Sample *op* includes copy operations and ϕ operations. The disjunction allows high confidence from any right-hand-side variable to be propagated to the left-hand-side variable and vice-versa. In our implementation, we generate different data-flow constraints depending on the different operand/result types (e.g., an addition of a floating point value and an integer value yields a floating point value). We omit the details as they are standard.

Our analysis is piggy-backed on a standard points-to analysis. To avoid bogus data flow, we only consider must-alias relations. Note that missing data flow is less critical to our analysis compared to others as it only means the corresponding type hints of the disconnected variables cannot be linked to acquire higher confidence. In many cases, the disconnected variables already have strong enough local type hints as suggested by our results in Section 6.

Example. We use an example in Fig. 8 to illustrate how to construct data flow constraints. The left side of the figure presents the program in *SSA* form, and the right side presents the corresponding constraints. In the *SSA* form, we use the superscripts to denote the different definitions of a variable. For example, the three definitions of z are denoted by z^1, z^2 and z^3, respectively. Observe at line 7, the type predicates for z^1 and z^2 are disjointed.

During the analysis, we also collect all the observable types involved in the project, which constitute the *type domain*. Later, for each type τ in the domain, our engine infers the probabilities of each variable having the τ type.

5.2 Attribute Constraints

Attribute constraints are extracted from attribute accesses, which provide type hints for the objects. Intuitively, the more (unique) attribute accesses we observe about an object, the better chance we can infer its type. Generating attribute constraints entails addressing the following challenges:

(1) *Path Sensitivity.* The type of a Python program variable may be path sensitive, meaning that the variable may have different types along different paths. As discussed in Section 2, if we do not carefully handle path sensitivity, the attribute constraints may become contradictory and lead to inference failure. This suggests that our attribute constraint construction shall be path-sensitive.

(2) *Dynamic Update of Attribute Set.* In Python, the attribute set of an object can be dynamically updated. Consequently, the relation between the type of an object and its attribute set may become uncertain. Here, we consider that dynamic updates do not change the type of an object, which is determined upon the object creation. Fortunately, probabilistic analysis allows us to naturally model such uncertainty. More details will be provided later in this Section.

(3) *Incompleteness in Attribute Accesses.* Attribute accesses provide a lot of type hints. However in most cases, we can only observe a subset of the attributes of an object. In other words, our observation is incomplete. As a result, there is uncertainty in such type hints even though they do provide useful information towards resolving the type. In our motivating example (Fig. 1), both types `Response` and `Request` have the attributes "*url*" and "*method*". The attribute accesses allow us to narrow down the set of possible types. But there is still uncertainty in deciding the type.

Handling Path Sensitivity (Challenge 1). To address the first challenge, we collect the attribute accesses of each variable x path-sensitively. The collection procedure consists of two steps: (1) Traverse the individual paths in the control flow graph of the procedure in which x resides and collect the attribute accesses of x for each path. Cycles are broken by unrolling each loop once. Note that the number of unrolling for a loop is irrelevant in our context. (2) Merge paths with the same set of attribute accesses and remove those that have an empty set. Note that if multiple paths have the same set of attribute accesses, they do not provide additional confidence of the inferred type as the confidence is derived from the uniqueness of the set of attribute accesses (i.e., how many types contain these attributes) instead of the number of paths along which these accesses are observed. Consider the example in Fig. 2. Since there are two paths in the program, we collect two sets of attribute accesses, {"*data*"} and {"*request*"}, for variable `r`.

After collecting the attribute accesses of x along each path, we then construct the attribute constraints of x. For each path i, we introduce a predicate $\mathcal{P}(x_i,\tau)$ to assert that the type of variable x in path i be τ. These constraints often have the following form.

$$\mathcal{P}(x_i,\tau) \xrightarrow{p} \{\text{attributes on } i\} \subset \mathcal{A}(\tau) \tag{10}$$

$$\{\text{attributes on } i\} \subset \mathcal{A}(\tau) = 1(p_0) \tag{11}$$

$$\{\text{attributes on } i\} \subset \mathcal{A}(\tau) \xrightarrow{p'} \mathcal{P}(x_i,\tau) \tag{12}$$

Intuitively, the first constraint asserts that if x_i is of type τ, the attributes observed on i are a subset of the attributes of type τ with probability $p = HIGH$ (i.e., 0.95). Note that using a close to 1.0 value instead of 1.0 to denote a very high probability is standard in probabilistic inference [12].

Handling Dynamic Attribute Updates (Challenge 2). Ideally, if the observed attributes are indeed a subset of the attribute set of τ, denoted as $\mathcal{A}(\tau)$, the boolean variable denoting $\{\text{attributes on } i\} \subset \mathcal{A}(\tau)$ has the true value, false otherwise. However in practice, if the attributes are not a subset of τ's attributes (according to τ's definition/original prototype), we cannot simply set the boolean variable to false as this may be caused by dynamic attribute updates. For instance, assume a new attribute a' is added to x and then accessed. The observed attributes are hence not a subset of τ's attribute set although the type of x is still τ. To model dynamic updates, ideally, we would identify all attribute updates (especially attribute additions) and the variables that are affected by these updates, and then change the attribute constraints according to the updates. In the aforementioned example, a' should be precluded from the observed attribute set of the constraint. However, achieving soundness in such analysis requires a full-fledged path-sensitive analysis. Our solution is hence to use probability to model the uncertainty caused by dynamic updates. Specifically, when the observed attribute accesses are not a subset of τ's attribute set, we assign a probability p_0 to the predicate (Equation 11), depending on the number of observed attributes that are part of τ's attribute set. The computation of p_0 is represented by the following formula.

$$p_0 = LOW + \frac{|\{\text{attributes on } i\} \cap \mathcal{A}(\tau)|}{|\{\text{attributes on } i\}|} \times (HIGH - LOW) \tag{13}$$

Note that if all the observed attributes are part of τ's attribute set, $p_0 = HIGH$. If none of the observed attributes are part of τ's attribute set, $p_0 = LOW$ (i.e., 0.05). An example will be presented later in this section. □

Handling Incompleteness in Attribute Accesses (Challenge 3). Equation 12 asserts given that the attributes observed on i are a subset of the attributes of type τ, x_i is of type τ with probability p'. The probability p' is determined by the uniqueness of the observed attributes, that is, how many other types also contain these attributes. The computation is represented by the following formula.

$$p' = 0.5 + 0.5 \times (2 \times HIGH - 1) \times \frac{1}{N} \tag{14}$$

N represents the number of types in the observable domain which contain the attributes that we observed. When N is very large, p' is close to 0.5, meaning that we are completely uncertain if x_i is of type τ or not. Note that $p' > 0.5$ means that we are positive about the assertion and $p' < 0.5$ means that we are negative (i.e., there are evidence for x_i not having type τ). When $N = 1$, $p' = HIGH$, meaning that we are highly confident that x_i is of type τ when the observed attributes are unique.

Computing the Probability at the Definition Point. Intuitively, if x has type τ at path i, it also has type τ at its definition point. Formally, if the attribute accesses of x can be collected from multiple paths, we use the maximal probability of all paths to represent the one at its definition point. The computation is represented as the following formula.

$$p(\boldsymbol{x}) = MAX(p(\boldsymbol{x}_1), p(\boldsymbol{x}_2), ..., p(\boldsymbol{x}_n)) \tag{15}$$

```
#Type τ has attributes a1, a2, a3, and a4 in its definition.
#Types τ1 and τ2 have attributes a1 and a2 in its definition.
#x is of type τ.
1: def foo(...):
2:   x = ...
3:   if ...:
4:     ... x.a1
5:   else:
6:     ... x.a2
7:   gee(x)
8:   if ...:
9:     ... x.a3
10: def gee(t):
11:   if ...:
12:     ... t.a2
13:   else:
14:     ... t.a4
15:   t.a5=... #attr. add
16:   if ...:
17:     ... t.a5
```

Figure 9: Example code for attribute constraints.

path	constraint
(1) TT in `foo()`	(a) $\mathcal{P}(x_1, \tau) \xrightarrow{0.95} \{a1, a3\} \subset \mathcal{A}(\tau)$ (b) $\{a1, a3\} \subset \mathcal{A}(\tau) = 1(0.95)$ (c) $\{a1, a3\} \subset \mathcal{A}(\tau) \xrightarrow{0.95} \mathcal{P}(x_1, \tau)$
(2) TF in `foo()`	(d) $\mathcal{P}(x_2, \tau) \xrightarrow{0.95} \{a1\} \subset \mathcal{A}(\tau)$ (e) $\{a1\} \subset \mathcal{A}(\tau) = 1(0.95)$ (f) $\{a1\} \subset \mathcal{A}(\tau) \xrightarrow{0.65} \mathcal{P}(x_2, \tau)$
(3) FT in `foo()`	...
(4) FF in `foo()`	...
(5) TT in `gee()`	(g) $\mathcal{P}(t_1, \tau) \xrightarrow{0.95} \{a2, a5\} \subset \mathcal{A}(\tau)$ (h) $\{a2, a5\} \subset \mathcal{A}(\tau) = 1(0.5)$ (i) $\{a2, a5\} \subset \mathcal{A}(\tau) \xrightarrow{0.65} \mathcal{P}(t_1, \tau)$
...	...

Figure 10: Attribute constraints for individual paths of the example in Fig. 9. "TT in `foo()`" means a path in `foo()` in which both predicates take the true branch.

$p(\boldsymbol{x})$ represents the probability of x having type τ at its definition point. Note that the probability $p(\boldsymbol{x}_i)$ of each path i can be computed by formulas stated in Section 4.

The Essence of Path Sensitivity in Our Technique. We cannot afford full-fledged path-sensitivity in general due to the exponential number of possible paths. We only consider path sensitivity of attribute accesses for the same variable (within the same procedure). Specifically, for each variable x in a procedure, we collect its attribute accesses path-sensitively within the procedure. Note that x may be related to other variables. The attribute accesses for those variables are collected independently and also in a path sensitive fashion. The correlations between x and these variables are modeled by the data flow constraints discussed in Section 5.1 that are path insensitive. From the results in Section 6.2, such a design allows us to achieve a balance between overhead and effectiveness.

Example. Consider the example in Fig. 9. Variable x is of type τ that has attributes `a1`, `a2`, `a3`, and `a4`. However this is unknown and we want to infer x's type from the code. Variable x is defined at line 2 and its attributes `a1`, `a2`, and `a3` are accessed at lines 4, 6 and 9, respectively. It is also passed to function `gee()` in which its attributes are accessed. A new attribute `a5` is added to the object at line 15.

Ideally, we would explore the individual whole program paths to construct attribute constraints. For example, a path $2 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 11 \rightarrow 14 \rightarrow 15 \rightarrow 16 \rightarrow 17 \rightarrow 8 \rightarrow 9$ contains accesses to attributes `a1`, `a4`, `a5`, and `a3`. Such whole-program path-sensitive analysis incurs prohibitive overhead. Instead, we collect the attributes for variable x in `foo()` along the intra-procedural path $2 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 9$ and the attributes for t in `gee()` along $11 \rightarrow 14 \rightarrow 15 \rightarrow 16 \rightarrow 17$. The former has attributes `a1` and `a3` whereas the latter has `a4` and `a5`. The likelihoods of x having type τ and t having type τ are first inferred independently based on the intra-procedural attribute

```
1: s = f(*args, **kwargs)
2: if isinstance(s, basestring):
3:   y = s.lower()
4: else:
5:   y = str(s)
```

Figure 11: An example for subtype constraints

sets. They are further aggregated through the data-flow constraint between x and t.

Fig. 10 shows the attribute constraints for individual paths. The first four rows denote the attribute constraints for the four paths in `foo()`. In particular, constraint (a) means that if `x` is of type τ in path (1) (both predicates take the true branch), it is highly likely that the observed attributes `a1` and `a3` are part of the attributes of τ. Constraint (b) means that `a1` and `a3` are highly likely to be in the attribute set of τ. Constraint (c) asserts the reversion of (a). Note that since the observed attributes `a1` and `a3` are unique to τ, the probability on top of the arrow is *HIGH*. Constraints (d)-(f) are similar except that the probability in (f) is 0.65 due to the fact that the observed attribute `a1` along path 2 is not unique to τ. Note that τ_1 and τ_2 also have `a1`. The probability 0.65 is computed by Equation 14.

Constraints (g)-(i) are for a path in `gee()` with both predicates taking the true branch. Observe that constraint (h) denotes that even though `a2` and `a5` are not part of the attribute set of τ due to the dynamic attribute addition at line 15, we do not set the predicate $\{a2, a5\} \subset \mathcal{A}(\tau)$ to false. Instead, it has a probability 0.5 of holding a true value, according Equation 13.

According to equations in Section 4, we can infer the probability of x having type τ at paths (1), (2) and (5) is 0.91, 0.63 and 0.41, respectively. With Equation 15, we further infer the probabilities of x and t having τ at their definition points are both 0.91. Combining the data-flow constraints $\mathcal{P}(x,\tau) \overset{1.0}{\longleftrightarrow} \mathcal{P}(t,\tau)$, we have the final likelihood of x being of τ is 0.99. Similarly, the likelihood of x having τ_1 is 0.74.

5.3 Subtype Constraints

Subtype constraints are extracted from the type checking statement `isinstance`(x, τ), which implies the possible types of a variable. We also generate the subtype constraints path sensitively, which is similar to the generation of attribute constraints. We use a simple example in Fig. 11 to illustrate how we generate subtype constraints. In this example, line 1 is an external function call, so we do not know the type of its return value, and hence the type of variable `s`. Line 2 represents a subtype checking of `s`. Intuitively, we know that when the execution goes into the true branch (line 3), the type of `s` must be a subtype of `basestring`. On the other hand, when the false branch (line 5) is taken, it must not be a subtype of `basestring`. Accordingly, assume we are computing the likelihood of the type of variable `s` being τ, we can generate the subtype constraints as follows.

$$\begin{array}{c|l} \text{(a)} & \mathcal{P}(s_1,\tau) \overset{0.95}{\longleftrightarrow} (\tau \preccurlyeq \mathit{basestring}) \\ \hline \text{(b)} & \mathcal{P}(s_2,\tau) \overset{0.05}{\longleftrightarrow} (\tau \preccurlyeq \mathit{basestring}) \end{array}$$

Here, s_1 and s_2 represent variable s in the two branches, and $\preccurlyeq$ represents a subtype relation. Intuitively, (a) denotes that when observing τ is a subtype of *basestring* (e.g., `str`), we know the type of variable `s` at line 3 is very likely to be τ, and vice versa. Constraint (b) is similarly interpreted.

5.4 Naming Constraints

Our technique is based on probabilistic inference, which allows us to take into consideration uncertain type hints even from the natural language perspective. The intuition is that when programmers name a variable, they often follow some conventions implying its type. In our motivating example in Fig. 1 we observe that lots of variables are lexically similar to `response` (e.g. `r`, `resp`), and most of them are indeed of the `Response` type. Besides, we can also infer the type of a variable according to other natural language features, such as *part-of-speech* and *singular/plural* form of nouns. For instance, a variable with a name starting with a verb (e.g., `has_connected`) very likely represents a boolean variable. When a variable's name is plural (e.g., connections), it probably represents a collection (e.g., `list`). Such type hints are uncertain and hence cannot be leveraged by most existing data-flow based type inference techniques. To model uncertainty, we use probabilistic constraints to represent the implicit relations between variables and naming conventions. In the following, we first illustrate how we use machine learning to mine the implicit naming conventions, and then discuss how to generate the naming constraints.

Naming Convention Learning. The learning component takes three inputs: (1) the variable set; (2) the observed type domain; and (3) the (partial) mapping from variables to their types, and finally produces a variable name classifier and a variable type classifier. The process consists of three main steps, First, it lexically clusters variable names using *k-means*. Then, it extracts natural language features from each variable. Finally, it trains the classification model for each type in the domain.

Clustering Variable Names. The procedure of clustering variable names consists of three steps: (1) normalize the variables; (2) compute variable name similarities; (3) cluster the variables using k-means. Normalization of a variable x entails the following. First, we remove all the tail digits of x's name if any. Second, we identify all the terms of x's name via some commonly used separators (e.g.,"_" and capital letters). Third, we transform each term into its lower case. Finally, we concatenate them again using separator "_". For instance, variable `hasConnected2` is normalized to "has_connected". For the computation of lexical similarity of x_1 and x_2, we use the following formula.

$$sim(x_1, x_2) = \frac{|lcs(\tilde{x_1}, \tilde{x_2})| + |las(\tilde{x_1}, \tilde{x_2})|}{2 \times min(|\tilde{x_1}|, |\tilde{x_2}|)} \qquad (16)$$

Here, $|lcs(\tilde{x_1}, \tilde{x_2})|$ denotes the length of the longest common substring between $\tilde{x_1}$ and $\tilde{x_2}$, which denote the normalized x_1 and x_2, respectively. $|las(\tilde{x_1}, \tilde{x_2})|$ represents the length of the longest common abbreviation between $\tilde{x_1}$ and $\tilde{x_2}$. We use the longest common abbreviation in addition to the longest common substring because we observe that abbreviations are widely used. For example, programmers may use a abbreviation "`pgm`" to represent the word "`program`". However, if we only use the longest common substring, the similarity between the two is low. In contrast, when the common abbreviation is considered, the similarity becomes much higher. For clustering variable names, we use k-means, which produces a classifier that can identify the variable name cluster of a given variable. The classifier is then used in the feature extraction for variables.

Extracting Natural Language Features. We extract four kinds of features from each variable. Given a variable x, we first compute the cluster id of this variable using the

aforementioned classifier. Second, we extract the *part-of-speech* features of x. We first split the normalized name of x into terms and then count the frequencies of the verbs and nouns as features. Third, we extract the *singular/plural* form feature of x's name. We split the normalized x into terms. If the terms contain a plural word, we set the feature to 1 , otherwise 0. Finally, we extract the lexical similarity between the normalized x and each normalized name of type τ in the domain. We extract this feature because we observe that a lot of variables are lexically similar to its type name.

Training Classification Models. We train a model for each type in the observed domain. It is a standard supervised machine learning procedure. Specifically, given type τ, we first extract the aforementioned four kinds of features for each variable x and set the target label as 1 when the inferred types of x contain τ (0 otherwise). We put all the data into a SVM classifier and train a model for τ, denoted as $\mathcal{M}(\tau)$. With the trained model $\mathcal{M}(\tau)$, if we have a new variable x', we can predict the probability of the type of x' being τ. The predicted probability will be used in our naming constraints.

Constructing Naming Constraints. As we will show in Section 6, due to the uncertainty of naming conventions, the probabilities produced by the models alone cannot effectively predict correct types. Instead, we represent them as naming constraints that are used together with other constraints to infer types. To construct naming constraints, we introduce a threshold η to represent the weight of the naming conventions in type inference. Basically, for a given type τ and a variable x, we have the following constraint:

$$\mathcal{N}(x,\tau) \xrightarrow{\eta} \mathcal{P}(x,\tau) \tag{17}$$

$\mathcal{N}(x,\tau)$ represents that the type of x can be τ according to the naming model $\mathcal{M}(\tau)$. In our motivating example, we set the threshold $\eta = 0.7$ (see $\mathcal{C}_5$ and $\mathcal{C}_6$ in Fig. 6). Note that we set the initial probability of $\mathcal{N}(x,\tau)$ to the predicted probability $\mathcal{M}(\tau)$. For example, we assume that the initial probability as 0.8 in our motivating example (see $\mathcal{C}_8$ in Fig. 6). Due to the space limitations, we move the details to our technical report [34].

6. EVALUATION

We have implemented a prototype in Python, which consists of three components, namely, the naming convention learning component, the constraint generator and the inference engine. The naming convention component relies on a widely used type inference engine *PySonar2* [1] to infer variable types. The machine learning subcomponents are implemented using *sklearn* [3]. The probabilistic inference engine is built on *pgmpy* [2]. We make our tool along with the evaluation environment publicly available at [34].

Our evaluation aims to address four research questions:

RQ1: How effective is our approach in inferring variable types compared to existing static analysis?

RQ2: How efficient is our approach? Can it scale to real-world Python programs?

RQ3: What is the impact of the thresholds?

RQ4: What is the impact of each kind of constraints?

6.1 Experiment Setup

We use *PySonar2*, a static Python type inference engine, as both the baseline for comparison and a building block for the naming convention component. *PySonar2* is based on abstract interpretation [1], an advanced static analysis technique. It was previously used by Google Inc. and its underlying analysis engine is currently used by *SourceGraph* [4], a widely used analysis system for code repositories. We could not compare with other research prototypes as they are not publicly available or cannot be properly installed.

Preparation. We extract variables whose types cannot be inferred by *PySonar2*. To collect the ground truth for these variables, we developed a dynamic analysis collecting the runtime types of variables by executing all the test cases of the benchmarks. The variables that are typed dynamically but not statically are the targets of our experiment. In other words, we want to observe how many of them can be typed by our technique and how precise the inferred types are compared to the observed types. Note that the dynamic analysis is only for the experiment, our technique is static.

Benchmarks. We select a set of real-world Python projects, as shown in Table 2 (columns 1-2). Most of them are from GitHub and widely used in practice. When choosing these projects, we also considered diversity. Observe that some of them are large, with the largest over 54K lines of Python code. These projects mainly fall into the following domains:

`Httpbin`, `httpie`, `requests` and `urllib3` - popular HTTP libraries and applications.

`Paramiko` - a widely used SSHv2 library.

`Fabric` - a well known remote deployment and system administration application.

`Bs4`, `pyquery` and `simplejson` - popular HTML/XML parsing libraries.

`Bottle`, `cherrypy`, `flask`, `tornado` and `web2py` - widely used web application and development frameworks.

`Pyspider` - a famous web crawling toolkit.

`Click` - a well known command line utility.

`GitPython` - a popular library used to interact with Git.

Computation of Recall and Precision. Before describing the results, we first explain how we evaluate the effectiveness. As described in Section 3, our system essentially computes an ordered list of types with probabilities. When the computed probability of a type is less than 0.5, we say the variable unlikely has that type. For types whose probabilities are over 0.5, we cluster them by a threshold GAP. Intuitively, we traverse the ranked type list (of a variable) and compute the gap g of probabilities between any two adjacent types τ_1 and τ_2. If $g > GAP$, we introduce a new cluster starting with τ_2. Eventually, we partition the list to clusters with the gap larger than GAP. Our tool will only report the first cluster. In practice, the first clusters may also be large. To avoid overloading the users, we introduce another threshold $TOPn$, which defines the maximum number of types the system reports.

We leverage the dynamically collected types to compute recall for variable x, denoted as $R(x)$, in the following.

$$R_x = \frac{|\mathcal{D}(x) \wedge \mathcal{C}_1(x, TOPn)|}{|\mathcal{D}(x)|} \tag{18}$$

$\mathcal{D}(x)$ represents the dynamically collected type set of variable x. $\mathcal{C}_1(x, TOPn)$ represents the reported type cluster of x, whose size is bounded by $TOPn$.

For precision $P(x)$, we use the following.

$$P_x = \frac{|Feasible(\mathcal{C}_1(x, TOPn))|}{|\mathcal{C}_1(x, TOPn)|} \tag{19}$$

Table 2: Summary of the experiment results.

Benchmarks		Var(#/%)	Positives	Recall(%)			Precision(%)			Type Set Size(Avg.)			Time(s)
Name	SLOC		(%)	Top 3	Top 5	Top 7	Top 3	Top 5	Top 7	Top 3	Top 5	Top 7	
httpbin	744	150/36.82	97.33	85.62	90.41	90.41	90.05	88.86	88.35	1	2	2	0.1557
httpie	2243	424/47.71	95.99	77.11	80.88	81.12	85.34	83.88	82.48	2	2	3	0.4473
paramiko	8267	1517/38.65	96.11	86.08	87.58	88.20	84.60	82.05	80.77	2	3	3	0.2301
urllib3	3112	820/59.28	97.44	80.05	82.04	82.66	83.34	81.23	80.09	2	3	3	0.7456
requests	10595	856/57.80	97.78	72.98	77.94	83.40	83.19	80.90	79.38	2	3	3	0.8125
fabric	3061	832/44.39	97.84	77.64	80.53	82.41	85.30	83.80	82.46	2	2	3	0.2211
bs4	3341	1202/68.41	97.09	72.13	76.08	78.47	80.48	77.31	75.65	2	3	4	0.9185
simplejson	4104	299/55.12	95.32	75.73	79.29	78.62	88.58	87.18	86.08	1	2	2	0.375
pyquery	1133	412/53.18	96.60	70.14	70.39	72.03	83.94	80.94	79.65	2	3	3	0.5734
bottle	2569	643/49.56	97.05	77.08	79.79	80.29	85.36	83.55	82.31	2	2	3	0.1454
cherrypy	18795	1277/45.62	99.30	76.49	80.25	80.86	84.15	82.22	80.92	2	3	3	0.6275
flask	2658	893/61.39	96.98	71.29	73.83	74.24	85.00	82.59	81.39	2	2	3	0.4805
tornado	11134	3144/56.02	95.29	76.54	78.81	79.89	85.07	82.69	81.54	2	2	3	0.9221
web2py	54479	1931/43.89	92.18	69.62	72.72	74.65	82.78	80.70	79.18	2	3	3	1.0143
pyspider	7129	1612/49.20	97.08	72.49	75.62	76.19	86.81	84.67	83.64	2	2	3	1.4565
werkzeug	11009	3094/53.71	95.99	73.56	77.60	79.42	83.01	80.34	79.05	2	3	4	1.2348
click	3837	1069/52.34	99.35	78.12	82.03	83.22	85.83	83.82	82.81	2	2	3	0.5822
GitPython	5838	2179/54.44	98.16	74.00	77.75	79.56	86.46	84.81	83.56	2	2	3	0.8258
Average		8588/51.53	96.83	75.93	79.09	80.31	84.96	82.86	81.63	2	2	3	0.6538

$Feasible(\mathcal{C}_1(x, TOPn))$ represents the reported types that are feasible at runtime. Since we do not have the ground truth, for each possible cluster size ranging from 1 to $TOPn$, we randomly select 20 variables to estimate the overall average precision. For each variable, we use the following procedure to determine if a type τ in the reported type set is feasible. (1) When the variable is a parameter (to an external API function), we first check the documentation of the API to determine if τ is correct. If there is no documentation, we randomly create a value of τ and pass it as an input to the function leveraging existing unit tests. If the execution does not crash, we consider it feasible. (2) If a variable is assigned the return value of an API call, we first check documentation. If the documentation is not available, we replace the function call with a random value of τ type, and observe whether the execution will induce a type exception. Eventually, we take the average over the precision of all the sampled variables.

Our experiments were on an Intel i7-3770U machine with 16GB RAM, Ubuntu 14.04 and Python 2.7.6.

6.2 Experimental Results

Table 2 shows the summary of results. Column 3 (`Var#`) presents the number and percentage of the variables whose types are observed during dynamic runs but cannot be completely inferred by *PySonar2*. Observe that *PySonar2* fails to infer types for a large number of variables, ranging from hundreds to thousands. Note that we do not consider variables whose types are not observed during execution as we cannot acquire any ground truth for those variables.

Effectiveness Evaluation (RQ1). For the evaluation of effectiveness, we use the naming convention threshold $\eta = 0.7$ (described in Section 5.4), and the probability gap $GAP = 0.1$ (described in Section 6.1). Columns 4-13 present the results. Column 4 (`Positives`) presents the percentage of variables having at least one type whose probability is over 0.5. Observe that most variables have at least one positive type. Columns 5-7 present the results of recall. We choose three $TOPn$ configurations, namely, 3, 5, and 7. Observe that we have a high recall with an average 75.93% at top 3, 79.09% at top 5 and 80.31% at top 7. Also observe that the recall becomes higher along with the increase of $TOPn$. Columns 8-10 present the precision. Observe that we also have a high precision with an average 84.96% at top 3, 82.86% at top 5 and 81.63% at top 7. The precision decreases with the increase of $TOPn$. Columns 11-13 describe the average size of the reported (first) cluster. Observe that the size is reasonably small.

Analysis Time (RQ2). Column 14 presents the average time for inferring the type of a variable. Observe that most variables can be inferred in one second, with an average 0.6538 second. This suggests our technique is sufficiently fast to be used in IDE (for real-world Python programs).

Impact of Thresholds (RQ3). We study the impact of three thresholds, namely, GAP, $HIGH$ and η. We use the threshold $TOPn = 5$. Fig. 12(a) shows the impact of GAP. We evaluate four settings, namely, 0.05, 0.10, 0.15 and 0.20. Observe that the recall has a small increase with the increase of GAP, while the precision decreases a little bit. Fig. 12(b) shows the impact of $HIGH$. Observe that the impact on both recall and precision is negligible. Fig. 12(c) presents the impact of η. Observe that it has an impact similar to GAP but in the other direction.

Impact of Constraints (RQ4). We evaluate the impact of each kind of constraints by adding one kind at a time with the order of *attribute constraints*, *subtype constraints*, *data-flow constraints* and *naming constraints*. For precision, we only focus on variables having at least one positive type. Fig. 12(d) shows the results. Observe that each kind has positive contribution to the recall. Especially, the impact of naming constraints is significant. The precision overall has a marginal decrease with the additions of constraints.

Threats to Validity. Our evaluation is performed on a limited set of programs. It is possible that the performance of our technique may vary for other programs. The evaluation of precision is sample based and requires substantial manual efforts. Sampling errors and human errors may skew the results. Our technique depends on a set of parameters. Although we have studied a number of parameter settings, the study is incomplete due to the large parameter space.

7. RELATED WORK

Probabilistic Modeling. Recently, probabilistic graph models were applied to type inference of JavaScript. Raychev et al. [10] predicted JavaScript types by learning a probabilistic model from a large repository of programs ("big code"). Our technique is different from theirs. First, [10] assumes a large number of programs from which variable names and types can be learned. In contrast, our technique

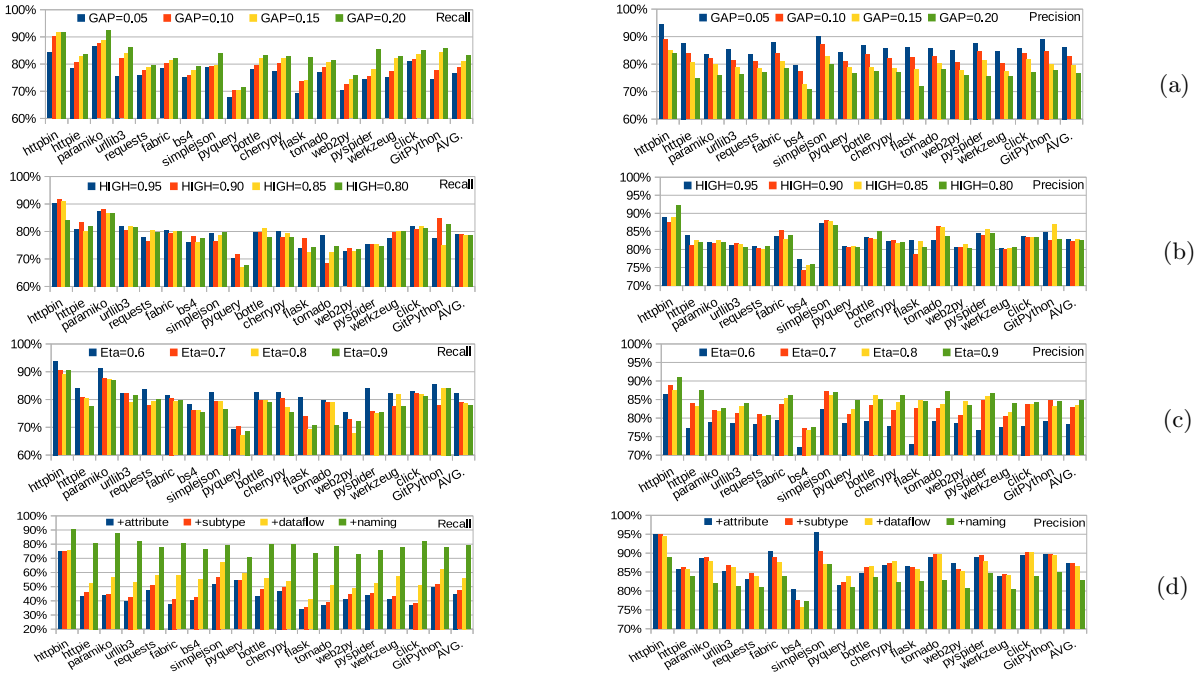

Figure 12: Impact of the thresholds (a-c) and each kind of constraints (d).

works on a single project. We do not assume a large code repository. Note that many Python types are project specific (30% by our experiment), only used in a project. Second, the model in [10] is monolithic and encodes properties of the entire program all together. A solution is global optimal which may sacrifice local optimality. In contrast, we analyze types of individual variables one by one, using separate models. Hence, our approach achieves local optimality, potentially having better precision and recall for individual variables. Note that during development, programmers likely query about types of individual variables instead of all variables. Third, we leverage project specific naming conventions, which have significant contribution to the performance according to our evaluation (Section 6). Probabilistic inference has also been widely used in many other areas such as software debugging [15], security [13], and specification extraction [11, 16, 12, 14]. Although we rely on a similar underlying inference engine, we address a set of unique challenges because of the different application domain.

Type Inference for Python. Several analyses have been developed for type inference of (a subset of) Python. Slib et al. [7] infer flow-insensitive types based on the Cartesian Product Algorithm (CPA). Cannon et al. [8] analyze atomic types from a local view of procedures. Aycock et al. [9] infer types for a subset of Python aggressively according to type consistency. Rigo et al. [5] and Gorbovitski et al. [6] optimize Python programs using abstract-interpretation based type inference. These approaches use forward analyses to infer variable types, and hence when encountering external function calls, they heavily rely on manual mocking. They may also fail to infer lots of types due to the lack of test drivers when analyzing libraries. Our approach is based on collecting type hints according to how variables are used, which requires much less mocking and test drivers. In addition, it leverages uncertain type hints from variable names and the reasoning engine is probabilistic inference.

Type Analysis for Other Dynamic Languages. Many type analyses have been proposed on other dynamic languages such as JavaScript [21, 22, 23, 24, 25], Ruby [26, 27, 28, 29] and PHP [30, 31]. TypeDevil [18] detects type inconsistencies in JavaScript according to dynamic observations of types. Their method is dynamic analysis based and depends on test drivers. Although their approach removes false positives using belief analysis, their belief is not a notion in probabilistic inference. DRuby [28] statically infers types for a subset of Ruby. It also starts from variables with known types. Rubydust [26] dynamically infers types in Ruby, and hence heavily depends on test coverage. Zhao et al. [31] developed a type inference technique for PHP optimizations. Ours is mainly for programming support and maintenance.

8. CONCLUSION

We propose a probabilistic inference based Python type inference technique. It allows us to leverage various type hints such as those derived from data flow, attribute accesses, and variable names. Some of them are uncertain. Our results show that our technique substantially outperforms a state-of-the-art type inference engine based on abstract interpretation. Our technique can type 79.09% of the variables that cannot be typed by abstract interpretation, with estimated 82.86% precision.

9. ACKNOWLEDGMENTS

This research was partially supported by DARPA under contract FA8650-15-C-7562, NSF under awards 1409668, 1320444, and 1320306, ONR under contract N000141410468, Cisco Systems under an unrestricted gift, China Scholarship Council (CSC) Scholarship, National Basic Research Program of China (973 Program) No.2014CB340702, National Natural Science Foundation of China No.91418202, No.61472178 and No.91318301. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

10. ARTIFACT DESCRIPTION

We package our system in a VMWare image which can be lauched on a host with VMWare Workstation (version 10) installed. In this section, we give detailed instruction about how to use the artifact.

Installation of VMWare Workstation. Our artifact VM requires VMWare workstation 10, which can be downloaded from [35]. Note that the Linux version of VMWare Workstation is packaged in a "`*.bundle`" executable so that one can use the command "`sudo sh /path/to/<filename>.bundle`" to install it.

Load the VM Image. First of all, go to the project website https://sites.google.com/site/pyprobatyping/ to download the VM image. After decompression, start VMware Workstation and go to menu `File>Open>Select the *.vmx file` to load the image.

Requirements for Starting the Image. The initial memory allocated for our VM image is 7GB, and the number of processors and cores is 2x2. If the host machine cannot support such a configuration, please go to `VM>Settings>Hardware Tab` to reset the resources before launching the VM. We have not tested our system with less than 7GB allocated memory. In addition, the image requires at least 15GB free disk space on the host. To get better performance, we would suggest to allocate more resources (e.g., memory and processors).

Artifact Contents. After loading the image, the HOME directory contains the following contents:

- A README file on the desktop (`$HOME/Desktop`) of the VM, including basic description of the working directory, execution requirements, instructions, evaluation and so on.
- The Python environment with our tool and the required libraries installed.
- The working directory (`$HOME/Current/NamingProject`), including benchmarks, data, test drivers and running scripts. The structure is listed as follows:
  - `Benchmarks` folder, including all the benchmark sources.
  - `Data` folder, including all the collected runtime data (e.g., variables and types) by our tracing tool.
  - `SData` folder, containing all the static data generated by PySonar2, which includes the statically inferred types of individual variables and data flow between variables.
  - `MData` folder, including all the merged data of dynamically and statically collected. This folder is initially empty and will be dynamically filled by our tool.
  - `tests` folder, including all the running scripts, configurations and test drivers.

We provide a number of case studies on our website https://sites.google.com/site/pyprobatyping/.

Running the Tool. First of all, open a terminal, and then move to the working directory by command: `cd $HOME/Current/NamingProject/tests`. We provide two kinds of evaluation. One is to execute all the benchmarks by command: `./run.sh [-l=<N>] <HIGH> <ETA>`. Here, `[-l=<N>]` is an option to choose how many kinds of constraints (`N=1...4`) are considered with the order corresponding to Fig. 12(d). Note that all kinds of constraints will be included if this option is omitted (i.e., `N=4` by default). `<HIGH>` and `<ETA>` represents the high probability threshold *HIGH* and the belief threshold η of the naming convention, respectively. For instance, to evaluate the results of Table 2, one needs to input the command `./run.sh 0.95 0.7`. For details of these thresholds, please refer to Sections 3 and 5.4. It will take several hours to complete all the benchmarks. Another choice is to run each benchmark one by one via command: `python run.py test-XX-YY [-l=<N>] <HIGH> <ETA>` where `XX` and `YY` represent the category and project name, respectively. One can find them in the form of `test-XX-YY.py` in the working test directory (`$HOME/Current/NamingProject/tests`). Take the benchmark `httpbin` as an example, the corresponding command is `python run.py test-httptools-httpbin 0.95 0.7`.

Evaluating the Results. The evaluation goal is to reproduce results shown in Table 2 and Figure 12. These are the main results of our experiment. To evaluate the results step by step, please follow the instructions in Section [Evaluation] of README. Our tool will output all the results in the folder `log/<project-name>`. The logged files are listed as below:

- `raw-diff-same-summaries.txt` stores the overall failure percentage of PySonar2. Note that the results are based on the traced variables.
- `analysis-summaries-<N>-<HIGH>-<ETA>.txt` stores partial (summarized) results corresponding to Table 2 and Figure 12.
- `analysis-results-<N>-<HIGH>-<ETA>.txt` presents the detailed results of the inferred types associated with probabilities. Refer to README for the meaning of each record.

11. REFERENCES

[1] PySonar2. https://github.com/yinwang0/pysonar2.

[2] pgmpy. http://pgmpy.org/.

[3] sklearn. http://scikit-learn.org/stable/.

[4] SourceGraph https://www.sourcegraph.com/

[5] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2006.

[6] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and K. T. Tekle. Alias analysis for optimization of dynamic languages. In *Dynamic Languages Symposium (DLS)*, 2010.

[7] M. Salib. Starkiller: A static type inferencer and compiler for python. In *Master's thesis, MIT*, 2004.

[8] B. Cannon. Localized type inference of atomic types in python. In *Master's thesis, California Polytechnic State University*, 2005.

[9] J. Aycock. Aggressive type inference. In *International Python Conference*, 2000.

[10] V. Raychev, M. Vechev and A. Krause. Predicting Program Properties from "Big Code". In *42th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2015

[11] N. E. Beckman and A. V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *32nd Annual Conference on Programming Language*

Design and Implementation (PLDI), 2011.

[12] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *30th Annual Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[13] Z. Lin, J. Rhee, C. Wu, X. Zhang, D. Xu DIMISUM: Discovering Semantic Data of Interest from Un-mappable Memory with Confidence. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, 2012.

[14] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[15] L. Dietz, V. Dallmeier, A. Zeller, and T. Scheffer. Localizing bugs in program executions with graphical models. In *23rd Annual Conference on Neural Information Processing Systems (NIPS)*, 2009.

[16] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[17] B. Moghaddam, T. Jebara, and A. Pentland. Bayesian face recognition. In *Pattern Recognition*, 2000.

[18] M. Pradel, P. Schuh, and K. Sen. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *37th International Conference on Software Engineering (ICSE)*, 2015.

[19] L. Gong, M. Pradel, M. Sridharan, and K. Sen. DLint: dynamically checking bad coding practices in JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[20] K. Sen, S. Kalasapur, T. Brutch, et al. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript In *9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013.

[21] W. Choi , S. Chandra, G. Necula, K. Sen SJS: A Type System for JavaScript with Fixed Object Layout In *22nd International Static Analysis Symposium (SAS)*, 2015.

[22] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *24th European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[23] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *16th International Static Analysis Symposium (SAS)*, 2009.

[24] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *19th European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[25] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium on Programming (ESOP)*, 2005.

[26] J. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for Ruby. In *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011.

[27] M. Furr, J. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2009.

[28] M. Furr, J. D. An, J. S. Foster, and M. W. Hicks. Static type inference for ruby. In *24th Annual ACM Symposium on Applied Computing (SAC)*, 2009.

[29] J. D. An, A. Chaudhuri, and J. S. Foster. Static Typing for Ruby on Rails. In *24th International Conference on Automated Software Engineering*, 2009.

[30] E. Kneuss, P. Suter, and V. Kuncak. Runtime instrumentation for precise flow-sensitive type analysis. In *1st International Conference on Runtime Verification*, 2010.

[31] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, et al. The hiphop compiler for php. In *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2012.

[32] O. Lhoták , L. Hendren. Scaling Java points-to analysis using SPARK, In *12th International Conference on Compiler Construction*, 2003.

[33] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations, *Exploring artificial intelligence in the new millennium 8*, 2003.

[34] PyProbaTyping. https://sites.google.com/site/pyprobatyping/.

[35] VMWare Workstation https://my.vmware.com/web/vmware/info?slug=desktop_end_user_computing/vmware_workstation/10_0