SpyType - Type-Based Abstract Semantics for Python
A Work in Progress

Andrei Nacu
"Alexandru Ioan Cuza" University, Faculty of Computer Science
Iasi, Romania

Python is a high level programming language that is strongly, but dynamically typed. In this paper we propose a type inference framework to compute types for the variables within a Python program by using static code analysis.

1 Introduction

Context and Motivation The present study introduces a framework that utilizes static code analysis to determine possible types for program variables within Python functions. Python is a dynamically typed programming language. This means that types are determined at runtime and are not explicitly declared. Our endgame is to create a transpiler between Python and C++ programs. We want our transpiler to be as similar to the Python source both in form and functionality. The main reason for this is that the developer will be familiarised enough with the code to be able to debug and fix potential issues.

C++ offers direct control over memory management and this may lead to better resource usage, if done correctly. Both programming languages have a large and experienced community. Python is one of the more popular choices for people who want to learn a new programming language. Python works best for scripting or prototyping different projects. But full-blown projects usually use C++ or other strongly and statically typed programming languages. Before too long, the necessity of implementing a proof of concept within a bigger solution will arise. Then, the prototype will need to be translated. And that is the moment where we want to help.

This paper tackles the problem of static type inference for Python programs. Our objective is to formalise and implement a prototype that is able to compute the possible variable types of a given Python function.

In our current state we have a work in progress prototype that is able to extract the Control Flow Graph (CFG) for a given Python function and perform data flow analysis on it [11]. We have a working framework that is able to perform type inference using test specifications for built-in functions and operators. We are currently looking into expanding the set of specifications in order to treat more cases, while testing the framework on more complex programs. We are currently using a minimal set of specifications for operators and built-in functions so we can test and gradually improve the prototype's scope and accuracy.

Contribution We propose a static analysis framework that is based on the abstract interpretation and dataflow analysis of Python code. Based on this formalisation we aim to create a prototype that is able to provide type information for variables for a given Python function. We base our inference mechanism on specifications extracted from the Python Language Reference [16].

For a given Python function received as input, our analysis has to provide the following information:

- the possible types of the input parameters and the return value, if any;
- the possible types of the local variables within the function;

Submitted to:
WPTE 2023

- the possible types of output parameters, if any.

In order to satisfy these requirements we introduce the following main concepts:

- type expressions, which are used to constrain type variables;
- abstract states, which combine the notions of type variables and type expressions to describe a program point. These states describe the possible types of variables in that specific point of the program, without losing information regarding the context in which types appear.

2 Related Work

Static Type Analysis by Abstract Interpretation of Python Programs. Raphaël Monat's ECOOP 2020 article [8], which was topped off by his PhD thesis, *Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries* [7], provides a very complex source of inspiration. The goal of this work is to catch possible runtime type errors in Python programs before the code is actually run. The proposed analysis is written using the MOPSA Project [9].

This work serves as an inspiration for what we aim to do. It is, however, context-sensitive and does not have the capacity to analyze functions in isolation. Our aim is to try and give as much information for a function without having to analyze the whole program. This is a very difficult task, but we believe that it is possible to achieve a good enough result for our purposes.

For example, the article states that `def f(a, b):  return a + b` is too risky to type infer on its own because of the fact that the + operator might be overloaded by the programmer. This is understandable. However, our angle is different. We presume that the code is written in a decent manner. As a first small step, we will consider that operator overloading does not take place and go with the built-in specifications. As a second step, we could expand this with the preliminary analysis of the overloaded operators in the program, calculate specifications based on them and add them to the already known built-in specifications.

MyPy. MyPy [10] is a static type checker for Python. It is a popular tool, it is used by many programmers and it is actively maintained. It also serves as an inspiration for the type system that we want to implement and has served as a starting point for other projects such as Nagini [5], which is a tool that is used to check the memory safety and can ensure data race freedom. MyPy is used mainly as a type checker for annotated functions. It can also be used for unannotated Python functions, but that is not the purpose of the project. The upside is that it can analyse isolated functions. The downside is that it does not provide information about the variables in every program point and it may not be able to infer the types of some variables. This is because it is a checker that verifies if the types of the variables are consistent with the types that are annotated.

For example, the following code snippet will be considered correct by MyPy:

```
def f(a) -> int:
    b = a + 5
    return b
x = f(3.5)
```

To be fair, there is no type error in this code. However, the type of the `f` function is annotated with `float` instead of `int` and this could make a difference when we want to translate the function declaration in C++.

Typeshed. Typeshed [15] is a collection of stubs that provide type annotations to the Python standard library and to some popular third-party libraries. The stubs follow the rules defined in the PEP483 Python

Enhancement Proposal [13]. It is used by many projects to check types for functions that are imported from these libraries. It is also used by IDEs like JetBrains PyCharm [14] or type checkers, like MyPy.

We are taking into consideration using the information from Typeshed in our own inference mechanism in the future. However, we have to pay attention to the fact that some type annotations may be too generic. For example, `def append(self, __object: _T) -> None: ...` from the `list` class does not provide any information about the fact that the function will add the type of the __object parameter to the list.

Pytype. The Pytype project is actively maintained and is even used by Google to type-check their programs. Its capabilities are not limited to type checking. It can also be used to detect typos or other common mistakes. By default, Pytype generates a *pyi* file that contains annotations for every function and variable within the code [2]. The annotations follow the guidelines imposed by PEP483 [13].

When it comes to functions, Pytype can be a little too generic. For example, if we provide the following as input:

```
def f(a, b):
  c = a + b
  return c
```

we get this output:

```
# (generated with --quick)
from typing import Any
def f(a, b) -> Any: ...
```

What we do is a little different. For a function with input parameters we try to infer their possible types based on the specifications of builtin operations and functions that use them as parameters.

Scalpel. Another worthy mention is the more recent Scalpel Framework [6]. It is a very interesting project that is worth keeping an eye on. It is a framework that is designed to run many types static analyses for Python programs, not just particularly type inference. It is at an early stage in development, but it is actively maintained.

Nuitka. Nuitka is a Python compiler that is written in Python [12]. Its objective is to provide a replacement for the Python interpreter. It translates Python code into C code and its aim is to compile a standalone executable, optimized for speed. Their builtin type inference is not yet matured, although there are discussions about it which we will have to keep in focus.

Codon. Codon, like Nuitka, is a very interesting Python compiler that focuses on performance [3]. It compiles Python into a standalone executable, but it does not yet work for all of Python's syntax. As far as type inference goes, heterogenous container support is still on the roadmap. It also accepts contributions, therefore we will keep it in focus, also.

3 Proposed Framework

We propose a framework that does type inference for program variables in Python functions. Program variable types are inferred based on specifications for the operations and functions that they are employed in. Each function call and operation adds a set of constraints to the possible types of program variables. The type inference problem is done by solving these constraints. Specifications for builtin operations and functions are introduced manually and are based on the Python Library Reference [16]. Dataflow analysis is used to gather constraints in every program point. The objective of dataflow analysis is to provide a fact for every program point, which is valid whenever that point is reached. Dataflow analysis problems are either forward or backward problems. For every program point, a forward problem computes facts based on its predecessor facts. On the other hand, a backward problem computes facts for a program point based on information held by the successors of that point. Our analysis is a forward problem. As per Nielson et al. [11], an instance of a dataflow problem consists of:

- a Control Flow Graph (CFG), that is a directed graph where each node represents a statement and each edge represents the control flow between statements;
- a complete lattice D, which is the domain of dataflow facts assigned to each program point;
- an initial dataflow fact, which specifies the fact that holds at the start of the program (because ours is a forward problem);
- a join operator that, for a CFG node, combines information from incoming edges;
- a dataflow transfer function $\varphi_i : D \rightarrow D$ for each node i in the CFG, which defines the effect of executing the statements in the node i.

The domain of dataflow facts is defined by *abstract states*. An abstract state assigns a possible type for each program variable, based on constraints gathered throughout code analysis. The basic building blocks of abstract states are type expressions. To represent program variable types we use a combination builtin Python types (like int, float, str, list and so on), sum types and type variables. Builtin Python types are generally used to represent cases where a program variable can be assigned a single type. Sum types are used to represent cases where a program variable must hold multiple types symultaneously and to describe elements inside an heterogenous container. Type variables are used in cases where the possible type of a program variable is a parametric type. The objective is to describe types that can be easily translated to C++. In C++, unions describe sum types and type variables are used in templates to enable parametric polymorphism. Our framework describes possible types for program variables through type expressions. The following notations are used to define type expressions:

- PT is the set of all builtin Python types;
- TV is the set of all type variables;
- $\oplus$ is the sum type operator. It is used to represent cases where a program variable can hold multiple types simultaneously. For example, int $\oplus$ float represents a type able to hold both int and float;

Definition 1. Type expressions *te* are defined by the following grammar:

$$c ::= \text{list} \mid \text{set} \mid \text{tuple}$$
$$te ::= pt \mid tv \mid te \oplus te \mid c\langle te\rangle \mid \text{dict}\langle te, te\rangle \mid \bot \mid \top$$

where $pt \in PT$ and $tv \in TV$.

The notation $c\langle te\rangle$ is used to describe generic containers. For example, list$\langle$int$\rangle$ describes an homogenous list that contains integers, while list$\langle$int $\oplus$ float$\rangle$ describes an heterogenous list that can contain both integers and floats (e.g. `[3, 3.5]`). In the notation dict$\langle te, te\rangle$ the first type expression describes the type of the keys and the second describes the type of the values. $\bot$ represents the bottom type. There are no cases where a program variable can be assigned the bottom type. This type is used to describe the type of an unitialized program variable or a type that cannot be inferred. $\top$ represents the top type, which describes every possible type. The top type is defined by the sum type of every possible type expression. Type variables are named using the syntax $T*$ (for example: T_1, T_a, $T?_1$ etc.).

Below are some examples in which type expressions are used to describe various variable types within a program:

Example 1.

```
a = 3
a = 3.5
```

Python allows for a variable to be assigned multiple types. The inferred type of `a` is $\text{int} \oplus \text{float}$ because it can hold both integer and float values.

Example 2.

```
a.pop()
a.append(3)
```

From the `pop` function call the inferred type of `a` is $\text{list}\langle T'_a \rangle$. The type variable T'_a represents the type of the elements in the list. This function call works for any type of element that is contained in `a`. After the `append` function call the inferred type of `a` becomes $\text{list}\langle T'_a \oplus \text{int} \rangle$. This type expression describes the fact that whatever the type contained by `a` was before the `append` call, it must now also hold integer values.

Over type expressions we define the partial order $\sqsubseteq$ as follows:

$$\frac{}{te \sqsubseteq te} \qquad \frac{}{te_i \sqsubseteq te_1 \oplus te_2}\ i = 1,2 \qquad \frac{te_i \sqsubseteq te'_i}{te_1 \oplus te_2 \sqsubseteq te'_1 \oplus te'_2}\ i = 1,2$$

$$\frac{}{\bot \sqsubseteq te} \qquad \frac{}{te \sqsubseteq \top} \qquad \frac{te \sqsubseteq te'}{c\langle te \rangle \sqsubseteq c\langle te' \rangle} \qquad \frac{te_1 \sqsubseteq te'_1, te_2 \sqsubseteq te'_2}{\text{dict}\langle te_1, te_2 \rangle \sqsubseteq \text{dict}\langle te'_1, te'_2 \rangle}$$

where te, te_i and te'_i are type expressions and $c\langle te \rangle$ represents a container type that has elements of the type given by the type expression te. For example, $\text{list}\langle \text{int} \oplus \text{float} \rangle$ is a list that contains elements that can be both integers and floats.

A type expression is deemed lesser than another if it provides more precise information. Note that not all type expressions are comparable. In certain instances, it may not be possible to assert that a particular type expression is more accurate than another:

- two different basic Python types are not comparable. For example, we cannot compare str and float;
- two different type variables are not comparable;

Below are some examples of comparable and incomparable type expressions:

- $\text{int} \oplus \text{float} \sqsubseteq \text{int} \oplus \text{float} \oplus \text{str}$;
- $\text{int} \oplus T_a \sqsubseteq \text{int} \oplus \text{float} \oplus T_a$;
- $T_a \sqsubseteq T_a \oplus T_b$;
- T_a and T_b are not comparable because they represent different type variables;
- $\text{int} \oplus \text{float}$ and $\text{int} \oplus \text{str}$ are not comparable because float and str are different basic Python types, which are not comparable.

Once the partial order $\sqsubseteq$ has been defined, it is possible to use the join and meet over type expressions. Here are several examples of how these operators are computed:

- $\text{int} \oplus \text{float} \oplus \text{str} \sqcap \text{int} \oplus \text{str} = \text{int} \oplus \text{str}$;
- $\text{int} \oplus \text{float} \oplus \text{str} \oplus T_b \sqcap \text{int} \oplus \text{str} \oplus T_b = \text{int} \oplus \text{str} \oplus T_b$;

- $\text{int} \sqcap \text{float} = \bot$;
- $T_a \oplus \text{int} \sqcap T_b \oplus \text{float} = \bot$;
- $\text{int} \oplus \text{float} \sqcup \text{int} \oplus \text{str} = \text{int} \oplus \text{float} \oplus \text{str}$;
- $\text{int} \oplus \text{float} \oplus T_b \sqcup \text{str} \oplus T_c = \text{int} \oplus \text{float} \oplus \text{str} \oplus T_b \oplus T_c$.

A *constraint over a type variable* is an expression of the form $tv \sqsubseteq te$, where tv is a type variable and te is a type expression. It defines the greatest type expression that a type variable may be substituted with. A set of constraints that need to be satisfied simultaneously over type variables describe a *context*. A context ctx is an expression of the following form:

$$ctx = tv_1 \sqsubseteq te_1 \wedge tv_2 \sqsubseteq te_2 \wedge \ldots \wedge tv_n \sqsubseteq te_n$$

Examples of contexts:

- $T_a \sqsubseteq \text{int} \oplus \text{float} \wedge T_b \sqsubseteq \text{str}$
- $T_a \sqsubseteq \text{int} \oplus \text{float} \oplus T_b \wedge T_a \sqsubseteq \text{float} \wedge T_b \sqsubseteq \text{int} \oplus \text{str} \equiv T_a \sqsubseteq \big((\text{int} \oplus \text{float} \oplus T_b) \sqcap (\text{float})\big) \wedge T_b \sqsubseteq \text{int} \oplus \text{str}$

Note that the constraint obtained by multiple constraints applied over the same type variable is equivalent to their greatest lower bound.

Solving a type variable in a context means finding a type expression that satisfies all constraints applied to that type variable. Constraints may have multiple solutions. The one that we keep is the least upper bound of all minimal solutions. Examples:

- for $T_a \sqsubseteq \text{str}$, our solution is $T_a = \text{str}$;
- for $T_a \sqsubseteq \text{int} \wedge T_a \sqsubseteq T_b$ we deduce $T_a = \text{int}$. This generates a new constraint: $\text{int} \sqsubseteq T_b$, which means that $T_b \sqsubseteq \text{int} \oplus T_b'$. This results in $T_b = \text{int} \oplus T_b'$, where T_b' is an unconstrained type variable;
- for $T_a \sqsubseteq \text{list}\langle \text{int} \oplus \text{float}\rangle \wedge T_a \sqsubseteq \text{list}\langle \text{int} \oplus T_b\rangle$ we deduce $T_a = \text{list}\langle \text{int} \oplus \text{float}\rangle$ and we generate a new constraint: $\text{float} \sqsubseteq T_b$. This means in $T_b \sqsubseteq \text{float} \oplus T_b'$, where T_b' is an unconstrained type variable;
- for $T_a \sqsubseteq \text{int} \oplus \text{float} \oplus \text{str}$, our solution is $T_a = \text{int} \oplus \text{float} \oplus \text{str}$. In this case, the minimal solutions are $T_a = \text{int}$, $T_a = \text{float}$ and $T_a = \text{str}$;
- for $T_a \sqsubseteq \text{int} \oplus \text{float} \oplus T_b \wedge T_b \sqsubseteq \text{str}$:
  - we know that $T_b \sqsubseteq \text{str}$, therefore $T_a \sqsubseteq \text{int} \oplus \text{float} \oplus \text{str}$. This results in $T_a = \text{int} \oplus \text{float} \oplus \text{str}$;
  - $T_b \sqsubseteq \text{str}$ results in $T_b = \text{str}$;
- for $T_a \sqsubseteq \text{int} \oplus \text{float} \oplus T_b \wedge T_b \sqsubseteq \text{str} \oplus T_a$:
  - we know that $T_b \sqsubseteq \text{str} \oplus T_a$, therefore $T_a \sqsubseteq \text{int} \oplus \text{float} \oplus \text{str} \oplus T_a$. The minimal solutions are $T_a = \text{int}$, $T_a = \text{float}$ and $T_a = \text{str}$. The least upper bound of these solutions is $T_a = \text{int} \oplus \text{float} \oplus \text{str}$;
  - we know that $T_a \sqsubseteq \text{int} \oplus \text{float} \oplus T_b$, therefore $T_b \sqsubseteq \text{str} \oplus \text{int} \oplus \text{float} \oplus T_b$. The minimal solutions are $T_b = \text{str}$, $T_b = \text{int}$ and $T_b = \text{float}$. The least upper bound of these solutions is $T_b = \text{str} \oplus \text{int} \oplus \text{float}$;

A *generic type variable assignment* for an analyzed Python function is an expression of the form:

$$ta = pv_1 : tv_1 \wedge pv_2 : tv_2 \wedge \ldots \wedge pv_n : tv_n$$

where pv_i iterate over all the program variables in the input function and tv_i are type variables. In a program point, an *abstract state* is described by one or more contexts alongside a generic type variable assignment. A abstract state $\widehat{\sigma}$ is an expression of the form:

$$\widehat{\sigma} = ta \wedge (ctx_1 \vee ctx_2 \vee \ldots \vee ctx_m)$$

where ctx_i are contexts. Examples of abstract states:

- $(\text{a}{:}T_a \wedge \text{b}{:}T_b) \wedge \big((T_a \sqsubseteq \text{int} \wedge T_b \sqsubseteq \text{int}) \vee (T_a \sqsubseteq \text{float} \wedge T_b \sqsubseteq \text{float})\big)$
- $(\text{a}{:}T_a \wedge \text{b}{:}T_b \wedge \text{c}{:}T_c) \wedge (T_a \sqsubseteq \text{int} \oplus \text{float} \wedge T_b \sqsubseteq \text{int} \wedge T_c \sqsubseteq \text{float})$

An abstract state $\widehat{\sigma}$ has a solution $\rho(\widehat{\sigma})$, which describes its *canonical form*:

$$\rho(\widehat{\sigma}) = pv_1{:}te_1 \wedge pv_2{:}te_2 \wedge \ldots \wedge pv_n{:}te_n$$

where pv_i are the program variables present in $\widehat{\sigma}$ and te_i are the solutions for the type variables assigned to pv_i in $\widehat{\sigma}$. The solution for a type variable in an abstract state is the sum of type expressions that describe solutions for that type variable in every context. Examples:

- for $\widehat{\sigma} = (\text{a}{:}T_a \wedge \text{b}{:}T_b) \wedge \big((T_a \sqsubseteq \text{int} \wedge T_b \sqsubseteq \text{int}) \vee (T_a \sqsubseteq \text{float} \wedge T_b \sqsubseteq \text{float})\big)$ the solution is $\rho(\widehat{\sigma}) = \text{a}{:}\text{int} \oplus \text{float} \wedge \text{b}{:}\text{int} \oplus \text{float}$
- for $\widehat{\sigma} = (\text{a}{:}T_a \wedge \text{b}{:}T_b \wedge \text{c}{:}T_c) \wedge (T_a \sqsubseteq \text{int} \oplus \text{float} \wedge T_b \sqsubseteq \text{int} \wedge T_c \sqsubseteq \text{float})$ the solution is $\rho(\widehat{\sigma}) = \text{a}{:}\text{int} \oplus \text{float} \wedge \text{b}{:}\text{int} \wedge \text{c}{:}\text{float}$

For two abstract states $\widehat{\sigma}_1$ and $\widehat{\sigma}_2$, $\widehat{\sigma}_1 \sqsubseteq \widehat{\sigma}_2$ if and only if $\rho(\widehat{\sigma}_1) \sqsubseteq \rho(\widehat{\sigma}_2)$. This means that, for every entry $pv{:}te_1$ in $\rho(\widehat{\sigma}_1)$, there exists $pv{:}te_2$ in $\rho(\widehat{\sigma}_2)$ such that $te_1 \sqsubseteq te_2$.

Two or more abstract states are *normalized* if the same program variable is assigned a type variable with the same identifier in all abstract states. For example:

- the following abstract states are normalized:

$$\begin{aligned}\widehat{\sigma}_1 &= (\text{a}{:}T_a \wedge \text{b}{:}T_b) \wedge (T_a \sqsubseteq \text{int} \wedge T_b \sqsubseteq \text{float})\\ \widehat{\sigma}_2 &= (\text{a}{:}T_a \wedge \text{c}{:}T_c) \wedge (T_a \sqsubseteq \text{str} \wedge T_c \sqsubseteq \text{list}\langle\text{int}\rangle)\\ \widehat{\sigma}_3 &= (\text{c}{:}T_c \wedge \text{d}{:}T_d) \wedge \big((T_c \sqsubseteq \text{float} \wedge T_d \sqsubseteq \text{str}) \vee (T_c \sqsubseteq \text{int})\big)\end{aligned}$$

- the following abstract states are not normalized:

$$\begin{aligned}\widehat{\sigma}_1 &= (\text{a}{:}T_a \wedge \text{b}{:}T_b) \wedge \big((T_a \sqsubseteq \text{int} \wedge T_b \sqsubseteq \text{int}) \vee (T_a \sqsubseteq \text{float} \wedge T_b \sqsubseteq \text{float})\big)\\ \widehat{\sigma}_2 &= (\text{a}{:}T_1 \wedge \text{b}{:}T_2) \wedge \big((T_1 \sqsubseteq \text{int} \wedge T_2 \sqsubseteq \text{int}) \vee (T_1 \sqsubseteq \text{float} \wedge T_2 \sqsubseteq \text{float})\big)\end{aligned}$$

Two or more abstract states can be normalized by renaming the type variables assigned to the same program variable with a common identifier. There must be no name clashes with existing type variable identifiers. A possible normalization for the two abstract states above is:

$$\begin{aligned}\widehat{\sigma}_1 &= (\text{a}{:}T'_a \wedge \text{b}{:}T'_b) \wedge \big((T'_a \sqsubseteq \text{int} \wedge T'_b \sqsubseteq \text{int}) \vee (T'_a \sqsubseteq \text{float} \wedge T'_b \sqsubseteq \text{float})\big)\\ \widehat{\sigma}_2 &= (\text{a}{:}T'_a \wedge \text{b}{:}T'_b) \wedge \big((T'_a \sqsubseteq \text{int} \wedge T'_b \sqsubseteq \text{int}) \vee (T'_a \sqsubseteq \text{float} \wedge T'_b \sqsubseteq \text{float})\big)\end{aligned}$$

Subsequently, we define the *join* and *meet* operations over normalized abstract states. Let $\widehat{\sigma}_1$ and $\widehat{\sigma}_2$ be two normalized abstract states:

$$\begin{aligned}\widehat{\sigma}_1 &= ta_1 \wedge (ctx_1 \vee ctx_2 \vee \ldots \vee ctx_m)\\ \widehat{\sigma}_2 &= ta_2 \wedge (ctx'_1 \vee ctx'_2 \vee \ldots \vee ctx'_k)\\ \widehat{\sigma}_1 \sqcup \widehat{\sigma}_2 &= (ta_1 \wedge ta_2) \wedge (\bigvee ctx \vee \bigvee ctx')\\ \widehat{\sigma}_1 \sqcap \widehat{\sigma}_2 &= (ta_1 \wedge ta_2) \wedge (\bigvee (ctx \wedge ctx'))\end{aligned}$$

where ctx and ctx' iterate over the contexts in $\widehat{\sigma}_1$ and $\widehat{\sigma}_2$ respectively. Observations:

- to join or meet two abstract states that are not normalized, their normalized form must be computed first;
- a conjunction $pv : tv \wedge pv : tv$ is equivalent to $pv : tv$ (i.e. $\mathrm{a}{:}T_a \wedge \mathrm{a}{:}T_a \equiv \mathrm{a}{:}T_a$). A similar rule applies for a conjuction between the same type variable constraints (i.e. $T_a \sqsubseteq \mathrm{int} \wedge T_a \sqsubseteq \mathrm{int} \equiv T_a \sqsubseteq \mathrm{int}$).

As previously stated, the domain of dataflow facts is described by abstract states. The *initial dataflow fact* in the analysis of a Python function is an abstract state where:

- all local variables are constrained by $\bot$. This denotes the fact that local variables are undefined at the beginning of the analysis. This includes the special variable `return` that represents the return value of the function;
- all function arguments are not constrained. We treat every function argument as a parametric type at the beginning of analysis.

For example, if a function `f(a, b)` contains the local variable `c` in its body, the initial dataflow fact is $\widehat{\sigma}_{init} = (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{c}{:}T_c \wedge \mathrm{return}{:}T_r) \wedge (T_c \sqsubseteq \bot \wedge T_r \sqsubseteq \bot)$.

Specifications for functions and operators are also given through abstract states. For example:

- the specification of a function f that takes an integer and a float as arguments and returns a float is:

$$spec_f = (\mathrm{p}_1{:}T_1 \wedge \mathrm{p}_2{:}T_2 \wedge \mathrm{return}{:}T_r) \wedge (T_1 \sqsubseteq \mathrm{int} \wedge T_2 \sqsubseteq \mathrm{float} \wedge T_r \sqsubseteq \mathrm{float})$$

- the specification of a function g that takes a list as input and modifies it by adding float elements to it:

$$spec_g = (\mathrm{p}_1{:}T_1) \wedge (T_1 \sqsubseteq \mathrm{list}\langle T_2 \oplus \mathrm{float}\rangle)$$

Note: operations are considered specific cases of functions, where the operands are formal parameters and the return value is the result of the operation.

Specifications are used to add constraints to an abstract state. Function calls and operators affect the possible types of operands and return values. To better illustrate this process we will take the following example:

Example 3.

- the current abstract state is: $\widehat{\sigma} = (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b) \wedge (T_a \sqsubseteq \mathrm{int} \oplus \mathrm{float} \wedge T_b \sqsubseteq \mathrm{int} \oplus \mathrm{float})$;
- the expression to be analyzed is `f(a, b)`;
- the specification for function `f` is $spec_f = (\mathrm{p}_1{:}T_1 \wedge \mathrm{p}_2{:}T_2 \wedge \mathrm{return}{:}T_r) \wedge (T_1 \sqsubseteq \mathrm{float} \wedge T_2 \sqsubseteq \mathrm{float} \wedge T_r \sqsubseteq \mathrm{float})$.

Applying a specification is done in the following manner:

- we obtain an intermediary abstract state $\widehat{\sigma}'$ by instantiating the function arguments with the actual arguments of the function call. This step substitutes:
  - the name of the program variables in the specification with the name of the actual arguments of the function call;
  - the type variables assigned to the program variables in the specification with the type variables assigned to the actual arguments of the function call. This affects both the generic type variable assignments and the contexts. Additionally, the `return` program variable is substituted with the expression that is the function call.

For our example we obtain: $\widehat{\sigma}' = (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{f}(\mathrm{a},\mathrm{b}){:}T_r) \wedge (T_a \sqsubseteq \mathrm{float} \wedge T_b \sqsubseteq \mathrm{float} \wedge T_r \sqsubseteq \mathrm{float})$

Note: $\widehat{\sigma}$ and $\widehat{\sigma}'$ are normalized abstract states.

- we compute the abstract state $\widehat{\sigma}_{expr}$ by computing the meet of the current abstract state $\widehat{\sigma}$ with the newly obtained intermediary abstract state $\widehat{\sigma}$'. This will add the constraints given by specifications to our input abstract state. If necessary, any newly added type variables are renamed so as to avoid name clashes between type variables.

  The abstract state obtained by applying $spec_f$ in our example is:

$$\begin{aligned}\widehat{\sigma}_{expr} &= \widehat{\sigma} \sqcap \widehat{\sigma}' \\ &= (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{f}(\mathrm{a},\mathrm{b}){:}T_r) \wedge (T_a \sqsubseteq \mathrm{int} \oplus \mathrm{float} \wedge T_a \sqsubseteq \mathrm{float} \wedge T_b \sqsubseteq \mathrm{int} \oplus \mathrm{float} \wedge T_b \sqsubseteq \mathrm{float} \wedge T_r \sqsubseteq \mathrm{float}) \\ &= (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{f}(\mathrm{a},\mathrm{b}){:}T_r) \wedge \big(T_a \sqsubseteq (\mathrm{int} \oplus \mathrm{float} \wedge \sqcap \mathrm{float}) \wedge T_b \sqsubseteq (\mathrm{int} \oplus \mathrm{float} \sqcap \mathrm{float}) \wedge T_r \sqsubseteq \mathrm{float}\big) \\ &= (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{f}(\mathrm{a},\mathrm{b}){:}T_r) \wedge (T_a \sqsubseteq \mathrm{float} \wedge T_b \sqsubseteq \mathrm{float} \wedge T_r \sqsubseteq \mathrm{float})\end{aligned}$$

  In this case, T_r is a new type variable obtained as a result of applying the specification.

The information contained by the nodes in the CFG corresponds to a Python line of code. A line of code usually contains a statement or expression. Every node has two program points: the entry of the node and the exit. The entry of a node is the program point that corresponds to the state of the program before executing the statement corresponding to the node. The exit of a node is the program point that corresponds to the state of the program after executing the statement.

The transfer function for a CFG node computes the abstract state at the exit of the node based on the abstract state at the entry of the node and the statement corresponding to the node. For different statement types, the transfer function for a CFG node i is defined as follows:

- if the information in node i is an assignment $pv = expr$:
  - we compute the abstract state $\widehat{\sigma}_{expr}$ that results from applying the specifications of the functions and operators involved in *expr* to the abstract state $\widehat{\sigma}^i_{in}$ at the entry of the node;
  - the intermediary abstract state $\widehat{\sigma}'_{out}$ is obtained from $\widehat{\sigma}_{expr}$ by assigning to *pv* the type variable assigned to the expression *expr* in $\widehat{\sigma}_{expr}$.
  - the exit abstract state $\widehat{\sigma}^i_{out}$ is computed from $\widehat{\sigma}'_{out}$ by removing generic type variable assignments that do not provide information for program variables and discarding all unused type variables.

  If the expression in Example 3 were modified to the statement `a = f(a, b)`, we would have:

$$\begin{aligned}\widehat{\sigma}_{expr} &= (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{f}(\mathrm{a},\mathrm{b}){:}T_r) \wedge (T_a \sqsubseteq \mathrm{float} \wedge T_b \sqsubseteq \mathrm{float} \wedge T_r \sqsubseteq \mathrm{float}) \\ \widehat{\sigma}'_{out} &= (\mathrm{a}{:}T_r \wedge \mathrm{b}{:}T_b \wedge \mathrm{f}(\mathrm{a},\mathrm{b}){:}T_r) \wedge (T_a \sqsubseteq \mathrm{float} \wedge T_b \sqsubseteq \mathrm{float} \wedge T_r \sqsubseteq \mathrm{float}) \\ \widehat{\sigma}^i_{out} &= (\mathrm{a}{:}T_r \wedge \mathrm{b}{:}T_b) \wedge (T_b \sqsubseteq \mathrm{float} \wedge T_r \sqsubseteq \mathrm{float})\end{aligned}$$

- if the information in node i is an assignment $pv = const$:
  - the intermediary abstract state $\widehat{\sigma}'$ is obtained from *const* using the output of the Python type function over the constant: $\widehat{\sigma}' = const{:}T_{const} \wedge \big(T_{const} \sqsubseteq \mathrm{type}(const)\big)$;
  - we compute the abstract state $\widehat{\sigma}_{const}$ that results from the meet operation between the input abstract state and $\widehat{\sigma}'$: $\widehat{\sigma}_{const} = \widehat{\sigma}^i_{in} \sqcap \widehat{\sigma}'$;
  - $\widehat{\sigma}^i_{out}$ is computed from $\widehat{\sigma}_{const}$ in the same fashion as for the assignment $pv = expr$

For example:

$$\begin{aligned}
\widehat{\sigma}^{i}_{in} &= (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b) \wedge (T_a \sqsubseteq \mathrm{float} \wedge T_b \sqsubseteq \mathrm{float}) \\
node_i &= (\mathrm{a} = 3) \\
\widehat{\sigma}' &= 3{:}T_{const} \wedge (T_{const} \sqsubseteq \mathrm{int}) \\
\widehat{\sigma}_{const} &= (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge 3{:}T_{const}) \wedge (T_a \sqsubseteq \mathrm{float} \wedge T_b \sqsubseteq \mathrm{float} \wedge T_{constr} \sqsubseteq \mathrm{int}) \\
\widehat{\sigma}^{i}_{out} &= (\mathrm{a}{:}T_{const} \wedge \mathrm{b}{:}T_b) \wedge (T_{const} \sqsubseteq \mathrm{int} \wedge T_b \sqsubseteq \mathrm{float})
\end{aligned}$$

- if the information in node i is just an expression *expr*, $\widehat{\sigma}^{i}_{out}$ is computed in the same way as for the assignment $pv = expr$, but without the step where the type variable assigned to *expr* is assigned to *pv* ($\widehat{\sigma}'_{out} = \widehat{\sigma}_{expr}$).

Having defined the basic notions of our type inference dataflow analysis framework, we will present some illustrative examples of how the analysis works.

Example 4.

```
def f():
    # initial input environment, all local variables are undefined
    ↓ $(\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{return}{:}T_r) \wedge (T_a \sqsubseteq \bot \wedge T_b \sqsubseteq \bot \wedge T_r \sqsubseteq \bot)$
    if random.randint(0,1) == 1:
        ↓ $(\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{return}{:}T_r) \wedge (T_a \sqsubseteq \bot \wedge T_b \sqsubseteq \bot \wedge T_r \sqsubseteq \bot)$
        a = 3
        ↓ $(\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{return}{:}T_r) \wedge (T_a \sqsubseteq \mathrm{int} \wedge T_b \sqsubseteq \bot \wedge T_r \sqsubseteq \bot)$
    # same input environment for all branches of the if-clause
    ↓ $(\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{return}{:}T_r) \wedge (T_a \sqsubseteq \bot \wedge T_b \sqsubseteq \bot \wedge T_r \sqsubseteq \bot)$
    else:
        ↓ $(\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{return}{:}T_r) \wedge (T_a \sqsubseteq \bot \wedge T_b \sqsubseteq \bot \wedge T_r \sqsubseteq \bot)$
        a = 3.5
        ↓ $(\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{return}{:}T_r) \wedge (T_a \sqsubseteq \mathrm{float} \wedge T_b \sqsubseteq \bot \wedge T_r \sqsubseteq \bot)$
    # gather information from both branches (join operation)
    ↓ $(\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{return}{:}T_r) \wedge \big((T_a \sqsubseteq \mathrm{int} \wedge T_b \sqsubseteq \bot \wedge T_r \sqsubseteq \bot) \vee (T_a \sqsubseteq \mathrm{float} \wedge T_b \sqsubseteq \bot \wedge T_r \sqsubseteq \bot)\big)$
    b = a + 10
    ↓ $(\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{return}{:}T_r) \wedge \big((T_a \sqsubseteq \mathrm{int} \wedge T_b \sqsubseteq \mathrm{int} \wedge T_r \sqsubseteq \bot) \vee (T_a \sqsubseteq \mathrm{float} \wedge T_b \sqsubseteq \mathrm{float} \wedge T_r \sqsubseteq \bot)\big)$
    return b
    ↓ $(\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{return}{:}T_b) \wedge \big((T_a \sqsubseteq \mathrm{int} \wedge T_b \sqsubseteq \mathrm{int}) \vee (T_a \sqsubseteq \mathrm{float} \wedge T_b \sqsubseteq \mathrm{float})\big)$
```

For this case we use the following specification for the + operator:

$$spec_{+} = (\mathrm{p}_1{:}T_1 \wedge \mathrm{p}_2{:}T_2 \wedge \mathrm{return}{:}T_3) \wedge \big((T_1 \sqsubseteq \mathrm{int} \wedge T_2 \sqsubseteq \mathrm{int} \wedge T_3 \sqsubseteq \mathrm{int}) \vee (T_1 \sqsubseteq \mathrm{float} \wedge T_2 \sqsubseteq \mathrm{float} \wedge T_3 \sqsubseteq \mathrm{float})\big)$$

This tells us that when the two parameters are integers, the return value of the operation is an integer. Similarly, when the two parameters are floats, the return value is a float.

Example 2

```
def g(a, b):
    # initial input environment, all local variables are undefined
    #  and also the input parameters are unconstrained
```

$$\downarrow\ (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{c}{:}T_c \wedge \mathrm{return}{:}T_r) \wedge (T_c \sqsubseteq \bot \wedge T_r \sqsubseteq \bot)$$

```
c = a + 3
```

$$\downarrow\ (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{c}{:}T_c \wedge \mathrm{return}{:}T_r) \wedge \big((T_a \sqsubseteq \mathrm{int} \wedge T_c \sqsubseteq \mathrm{int} \wedge T_r \sqsubseteq \bot) \vee (T_a \sqsubseteq \mathrm{float} \wedge T_c \sqsubseteq \mathrm{float} \wedge T_r \sqsubseteq \bot)\big)$$

```
b.append(c)
```

$$\downarrow\ (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{c}{:}T_c \wedge \mathrm{return}{:}T_r) \wedge \big((T_a \sqsubseteq \mathrm{int} \wedge T_c \sqsubseteq \mathrm{int} \wedge T_r \sqsubseteq \bot \wedge T_b \sqsubseteq \mathrm{list}\langle T_c \oplus T_b^c \rangle) \vee (T_a \sqsubseteq \mathrm{float} \wedge T_c \sqsubseteq \mathrm{float} \wedge T_r \sqsubseteq \bot \wedge T_b \sqsubseteq \mathrm{list}\langle T_c \oplus T_b^c \rangle)\big)$$

```
return c
```

$$\downarrow\ (\mathrm{a}{:}T_a \wedge \mathrm{b}{:}T_b \wedge \mathrm{c}{:}T_c \wedge \mathrm{return}{:}T_c) \wedge \big((T_a \sqsubseteq \mathrm{int} \wedge T_c \sqsubseteq \mathrm{int} \wedge T_b \sqsubseteq \mathrm{list}\langle T_c \oplus T_b^c \rangle) \vee (T_a \sqsubseteq \mathrm{float} \wedge T_c \sqsubseteq \mathrm{float} \wedge T_b \sqsubseteq \mathrm{list}\langle T_c \oplus T_b^c \rangle)\big)$$

For this example we use the following specification for the `append` function:

$$spec_{append} = (\mathrm{p}_1{:}T_1 \wedge \mathrm{p}_2{:}T_2) \wedge (T_1 \sqsubseteq \mathrm{list}\langle T_1^c \oplus T_2 \rangle)$$

Based on this, we are also able to compute a specification for g as follows:

$$spec_g = (\mathrm{p}_1{:}T_1 \wedge \mathrm{p}_2{:}T_2 \wedge \mathrm{return}{:}T_r) \wedge \big((T_1 \sqsubseteq \mathrm{int} \wedge T_2 \sqsubseteq \mathrm{list}\langle T_2^c \oplus T_r \rangle \wedge T_r \sqsubseteq \mathrm{int}) \vee (T_1 \sqsubseteq \mathrm{float} \wedge T_2 \sqsubseteq \mathrm{list}\langle T_2^c \oplus T_r \rangle \wedge T_r \sqsubseteq \mathrm{int})\big)$$

4 Conclusion and Future Work

Type inference for Python programs provides a way to better understand the source code. It also offers valuable information for porting or adapting code into statically typed programming languages. We proposed an inference system that is able to compute types by using dataflow analysis. We are currently working on a prototype that is able to infer types for every program point. Based on specifications for a limited set of builtin functions and operators, we managed to run analysis for a number of test functions. This is still a work in progress, but we believe that the results are promising. We aim to continue adding more specifications for widely used builtin functions and operators. We are also looking into real world modules for test cases. Additionally, more test cases will also help use add safeguards to detect situations where types cannot be inferred. We want to continue our research by finding a method to add support for recursive functions. Furthermore, we plan to extend the inference system to support more complex Python features, such as classes and decorators.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi & Jeffrey D. Ullman (2006): *Compilers: Principles, Techniques, and Tools (2nd Edition)*, pp. 624–636. Addison-Wesley Longman Publishing Co., Inc., USA. Available at `https://dl.acm.org/doi/10.5555/1177220`.

[2] Matthias Kramm et al. (Accessed on May 14, 2023): *Pytype: A Static Type Analyzer for Python*. `https://github.com/google/pytype`. GitHub repository.

[3] (Accessed on June 21, 2023): *Codon*. `https://docs.exaloop.io/`.

[4] coetaur0 (Accessed on May 13, 2023): *Staticfg: A Python Static Analysis Tool*. `https://github.com/coetaur0/staticfg`. GitHub repository.

[5] Marco Eilers & Peter Müller (2018): *Nagini: A Static Verifier for Python*. In Hana Chockler & Georg Weissenbacher, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 596–603, doi:10.1007/978-3-319-96145-3_33.

[6] Li Li, Jiawei Wang & Haowei Quan (2022): *Scalpel: The Python Static Analysis Framework*. *arXiv preprint arXiv:2202.11840*.

[7] Raphaël Monat (2021): *Static type and value analysis by abstract interpretation of Python programs with native C libraries*. Theses, Sorbonne Université. Available at `https://theses.hal.science/tel-03533030`.

[8] Raphaël Monat, Abdelraouf Ouadjaout & Antoine Miné (2020): *Static Type Analysis by Abstract Interpretation of Python Programs*. *Dagstuhl Artifacts Series* 6(2), pp. 11:1–11:6, doi:10.4230/DARTS.6.2.11. Available at `https://drops.dagstuhl.de/opus/volltexte/2020/13208`.

[9] (Accessed on June 5, 2023): *MOPSA Project*. `https://mopsa.lip6.fr/`.

[10] (Accessed on June 6, 2023): *MyPy: Optional Static Typing for Python*. `https://mypy-lang.org/`.

[11] Flemming Nielson, Hanne Riis Nielson & Chris Hankin (1999): *Principles of Program Analysis*, pp. 35–139. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-662-03811-6_2. Available at `https://doi.org/10.1007/978-3-662-03811-6_2`.

[12] (Accessed on June 21, 2023): *Nuitka the Python Compiler*. `https://nuitka.net/`.

[13] (Accessed on June 17, 2023): *PEP 483: The Theory of Type Hints*. `https://peps.python.org/pep-0483/`.

[14] (Accessed on May 13, 2023): *Type Hinting in PyCharm*. `https://www.jetbrains.com/pycharm/`.

[15] Python Typeshed Contributors (Accessed on June 6, 2023): *Typeshed: Collection of Type Hints for the Python Standard Library*. `https://github.com/python/typeshed`.

[16] Python Core Team: *Python 3 Documentation*. `https://docs.python.org/3/library/index.html`. Accessed on May 5, 2023.