Static Type Inference for Foreign Functions of Python

Mingzhe Hu, Yu Zhang*, Wenchao Huang, Yan Xiong
University of Science and Technology of China, Hefei, China
hmz18@mail.ustc.edu.cn, {yuzhang*, huangwc, yxiong}@ustc.edu.cn

***Abstract*—Static type inference is an effective way to maintain the safety of programs written in a dynamically typed language. However, foreign functions implemented in another programming language are often outside the inference range. Python, a popular dynamically typed language, has a lot of widely used packages which follow the multilingual structure with C/C++ extension modules. Existing deterministic Python static type inference tools which are not based on type annotations can do nothing about these foreign functions. In this paper, we propose a novel method to infer the type signature of foreign functions by analyzing implicit information in the layer of foreign function interface. We design a static type inference system, its evaluation on CPython, NumPy and Pillow shows that our method soundly infers the number and type of arguments for most foreign functions. Our results can further work as a complement to the state-of-the-art Python static type inference tool and enable it to analyze programs with foreign function calls. We catch 48 bugs of mismatch between foreign function declaration and its implementation, which make a parameter-free foreign function take argument of any type. 8 of the bugs we reported have been confirmed and fixed by communities.**

***Index Terms*—foreign function interface, static analysis, type system, static type inference, dynamically typed language**

I. Introduction

Python has become one of the most popular programming languages in the bloom of data science and machine learning, especially for its diverse libraries, many of which contain extension modules. Python front-end with C/C++ native implementation achieves both productivity and performance, almost becoming the standard structure for many mainstream software systems like NumPy, Pillow, TensorFlow, PyTorch, etc. However, feature discrepancies between two languages can pose many safety hazards in foreign function interface (FFI) for Python called Python/C API [1]. Type misuse is one of these common problems.

Many efforts have been made to design and implement static type inference methods and tools for Python, from both industry and academia. Mypy [2] from Python community, Pyright [3] from Microsoft, and Pyre [4] from Facebook are based on type annotations [5], which require modification of sources and violate the agile coding practice of Python. Allamanis *et al.* [6] and Xu *et al.* [7] propose machine learning based approaches which produce non-deterministic probabilistic results. Other representative tools which are deterministic and not based on type annotations do not support static type inference for foreign functions, including [8]–[10], Pytype [11] from Google, and PySonar [12] used by Google and SourceGraph. Without the ability to infer the type signatures of foreign functions automatically, these tools either ignore them which brings serious precision loss, or preset type stubs which requires heavy manual effort.

FFI is a mechanism by which a program written in one programming language can call routines or make use of services written in another [13]. A fully featured FFI can be divided into two layers, one is the machine dependent layer such as `libffi` [14], the other is the upper layer specifying call interface descriptions like type conversions for values passed between languages. For a statically typed language like Java or OCaml, foreign functions are declared with explicit types [15], [16]. However, it is not unsolvable for a dynamically typed language like Python. Our **key insight** is to statically analyze the implicit information within the call interface description, and infer constraints such as the number and type of arguments imposed on foreign functions.

We propose PyCType, a static type inference system for foreign functions of Python. PyCType is built on a set of rules to infer type signatures for foreign functions. These rules can be divided into three groups: (1) *Foreign function declaration* indicates how a foreign function called from the Python side is constructed with its C implementation and calling convention. (2) *Parameter type conversion* indicates how arguments passed from the Python side are converted to objects of C types, including parameter-free analysis based on calling convention flag, unused parameter analysis, and analysis of Python/C argument parsing APIs. (3) *Return type conversion* indicates how C variables are used to build the return value for a foreign function call from Python, which supports multiple returns. Rules on return type conversion comprise Python/C value building APIs analysis, Python/C explicit conversion APIs analysis, type cast analysis, and reaching definition analysis.

We evaluate PyCType on CPython, NumPy and Pillow, three representative and heavily used Python/C multilingual projects with a large amount of foreign functions. For 1751 foreign functions in total, we infer type signatures for 1338 (76.4%) of them. As our static type inference system is conservative and sound, the results are correct but may be imprecise. The return of a foreign function can be an argument of other foreign or local functions, making some degree of incompleteness and imprecision acceptable. Using type inference results as a complement to Pytype, the state-of-the-art Python static type inference tool, our system is proved to be effective by increasing recall precision by 27.5% on average for files from popular projects in different application domains.

To the best of our knowledge, this is the first study that (i) proposes a static type inference system for foreign functions of Python without using machine learning and type annotations and (ii) investigates deficiencies of mismatching foreign function implementation with calling convention declaration, which violate argument type requirements of foreign functions.

In summary, we make the following contributions:

- We define a static type inference system for foreign functions of Python, which is formulated by a series of rules. Our system comprises several static analyses of interface code between Python and C. Our method based on foreign function interface can be extended to dynamically typed host language other than Python.
- We prototype our method in PyCType[1], its evaluation on three representative projects shows that PyCType can infer type signature for most foreign functions. The inference results are sound as designed and the completeness can be improved by accessing more Python/C APIs into our system. Effectiveness of PyCType is demonstrated by using its results as a complement to Pytype, enabling it to analyze Python programs with foreign functions.
- As a part of our type system, rules for checking the consistency of foreign function declarations and implementations are denoted as gated semantic predicates [17] and their conjunctive normal form. Our system found 48 mismatch bugs from three commonly used projects. These bugs make it legal to pass argument of any type to a parameter-free foreign function, 8 of which have been confirmed and fixed by the communities.

II. Background

This section describes the architecture of a Python/C multilingual software system and the disability of Python static type inference from a single language view.

A. Python/C Multilingual Architecture

Multilingual software systems such as NumPy and Pillow are written in multiple languages. The host language Python aims to boost development productivity, and the foreign language C aims to support implementing high performance libraries. Between two languages is the interface code, which is C code using Python/C API. Fig. 1 shows a simplified example of C extension module to Python.

Lines 1–3 of Fig. 1(b) declare the method table of extension module `extm`, where each entry of type `PyMethodDef` describes a foreign function with four fields in turn: `ml_name` is the name of the method to be called from the Python side, `ml_meth` is the pointer to the C implementation, `ml_flags` is a bit field flag indicating how the call should be constructed, and optional `ml_doc` is the documentation string. As shown in line 2 of Fig.1(b), `"foo"` is the foreign function name called on the Python side (`extm.foo` in Fig. 1(a)), `_foo` points to the C function defined in Fig. 1(c), the bit field

[1]PyCType can be downloaded at https://github.com/S4Plus/pyctype

```
import extm
...
size = extm.foo(width, height)
```

(a) host.py

```
1 static PyMethodDef extMethods[] = {
2   {"foo", (PyCFunction)_foo, METH_VARARGS, "docstring"}
3 };
4 static struct PyModuleDef extModule = {
5   PyModuleDef_HEAD_INIT, "extm", "docstring", -1,
      extMethods
6 };
```

(b) interface.c

```
static PyObject* _foo(PyObject* self, PyObject* args) {
  int x, y;
  if (!PyArg_ParseTuple(args, "II", &x, &y))
    return NULL;
  return PyLong_FromLong(x * y));
}
```

(c) foreign.c

Fig. 1. An example of C extension module to Python.

flag `METH_VARARGS` indicates a typical calling convention, where C function implementation has the type `PyCFunction` that expects two `PyObject*` values. The first one is the module object when the method table is initialized with `PyModuleDef`, or `self` object for method when declared with `PyTypeObject` instead. The second parameter (often called `args`) is a tuple object packing all arguments passed from the Python side.

B. Static Type Inference for a Single Language

Existing deterministic Python static type inference tools which are not based on type annotations analyze programs from a single language view [8]–[12]. For a foreign function call like `extm.foo` in Fig. 1(a), neither the parameter type nor the return type can be inferred statically. Thus types of the arguments (`width` and `height`) cannot be checked at compile time, which makes it difficult to maintain the reliability.

This disability comes from the established assumption that type information cannot be accessed at compile time for dynamically typed languages. It seems true when we compare the foreign function declaration of Python in Fig. 1 with those in statically typed languages. For example, in Java, keyword `native` declares a foreign function named `bcopy`, which takes an argument of type `byte[]` and returns `void`.

```
private native void bcopy(byte[] arr);
```

In OCaml, a foreign function accepts two arguments: the name of the C function to bind and a value that describes the type of the bound function. The `@->` operator adds an argument type to the parameter list, and `returning` terminates the parameter list with the return type.

```
let newwin =
  foreign "newwin"
    (int @-> int @-> int @-> int @-> returning window)
```

Since large parts of Python standard libraries and built-ins are also C foreign functions, the state-of-the-art Python static type inference tool Pytype [11] from Google embeds Typeshed [18] from Python community as external type annotations. Typeshed and a similar module called `stubgen` inside Mypy [2] produce type stubs with manual coding, documentation extraction, and runtime introspection. These methods cannot be applied on a large scale, thus Typeshed only supports a few small third-party libraries. At the same time, C extension modules are often private libraries only for internal use in real world scenarios.

III. Key Insight Overview and Definitions

In this section, we formulate our key insight to model the problem of static type inference of Python foreign functions, as well as the definition of abstract syntax and types of a Python/C multilingual system.

A. Key Insight Overview

Without explicit type declarations, interface code of Python need indicate the call interface description to bridge different language features between two languages, *e.g.,* memory management [19], [20] and exception handling [21], [22]. We focus on type system in this paper by modelling and analyzing the implicit information of type conversion in the interface layer.

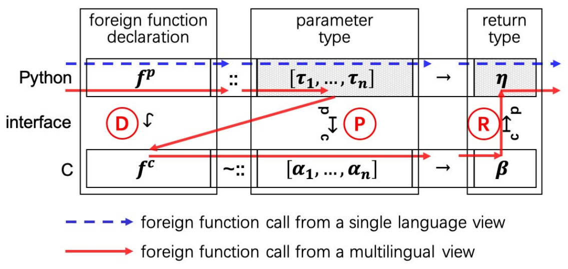

foreign function declaration
parameter type
return type
Python
interface
C
f^p
::
$[\tau_1, \ldots, \tau_n]$
→
η
D
P
R
p
c
f^c
~::
$[\alpha_1, \ldots, \alpha_n]$
β
foreign function call from a single language view
foreign function call from a multilingual view

Fig. 2. Foreign function call from a multilingual and type system view.

Fig. 2 shows the comparison of foreign function calls from single-language and multilingual type system views. From a single language view shown by the dashed blue line, the parameter types and return type of a foreign function cannot be inferred statically. Because unlike local function implemented in Python, foreign function is implemented in C with Python/C API. From a multilingual view shown by the solid red line, instead of explicit type declarations, interface code of a dynamically typed language indicates how to declare foreign functions (Ⓓ) and how to convert types forth (Ⓟ) and back (Ⓡ) via implicit information given by Python/C API and program properties.

To model and analyze these implicit information, we formulate the type inference rules with three premise parts shown in Fig. 2: foreign function declaration Ⓓ, parameter type conversion Ⓟ, and return type conversion Ⓡ. Let's take Fig. 1 as an example to illustrate these three types of premises:

- Premise Ⓓ is to map foreign function name f^{p} to its C implementation f^{c} via `PyMethodDef`, *i.e.,* mapping `"foo"` to `_foo` in Fig. 1. Note that f^{c} is the C function pointed to by the field `ml_meth` of the structure `PyMethodDef`, this function has type `PyCFunction` and takes two arguments `self` and `args` talked in Section II-A. So the parameter type and return type on the C side are not the signature of f^{c}, but the converted signature of f^{p} seen from the C side.
- Premise Ⓟ is to convert arguments from the Python side into C variables according to the type constraints hidden behind the format string `"II"` of Python/C API `PyArg_ParseTuple` function call. The format string `"II"` indicates that two arguments of Python integer should be passed to the foreign function and will be converted to C variables of type `unsigned int`.
- Premise Ⓡ is to convert C variables to Python types used as the return of a foreign function. In Fig. 1, Python/C API function `PyLong_FromLong` imposes a conversion from C integer to Python integer.

According to the above premise constraints, if a Python object other than an integer is passed to the foreign function, it will get a `TypeError` at runtime as Python uses dynamic type checking; if a Python integer with negative or zero value is passed, it will overflow the C variable and return with an unexpected value. This small example shows that static type inference for foreign functions of Python is necessary to maintain the reliability of such multilingual systems.

Hypothetical judgment forms and more static analysis patterns on Python/C API and C program properties will be formulated in detail in Section IV.

B. Abstract Syntax

We use the grammar shown in Fig. 3 to describe the abstract syntax of a Python/C multilingual system. To avoid ambiguity, we use superscript p and c to mark different language sides, *i.e.,* p for Python and c for C. The grammar is simplified with necessary language constructs, so that we can focus on type inference rules for foreign functions in this paper. PyCType fully supports CPython 3.3+ and complete C99 language.

On each language side, a program p is a statement list. Statements follow different syntax of their own language side. In Python, a local function is defined and applied both on the Python side, while a foreign function is declared and defined on the C side before it can be called on the Python side. A foreign function can be a module function or class method, which has no effect on inferring its type signature.

C. Types

The type sets of Python and C are denoted as $\mathbb{T}_{\mathrm{p}}$ and $\mathbb{T}_{\mathrm{c}}$ respectively. Fig.4 defines the abstract syntax forms of each Python type, *i.e.,* $\tau, \eta \in \mathbb{T}_{\mathrm{p}}$, and Fig.6 represents those of a C type, *i.e.,* $\alpha, \beta \in \mathbb{T}_{\mathrm{c}}$.

As a dynamically typed language, types on the Python side contains type values that can be bound to a Python variable when program is executed, like `str` (`unicode` in CPython 3.x), `int`, `object`, `dict`, etc. Besides these built-in types, product type ($pProduct$) can be used to represent types like `list`, `tuple` and `dict`, function type ($pFunc$) can be used to represent Python functions (local or foreign). $pUnion$ is

$$
\begin{array}{lcll}
p^{\mathrm{p}} & ::= & [s^{\mathrm{p}}_1, \ldots, s^{\mathrm{p}}_n] & \text{(Python program)} \\
s^{\mathrm{p}} & ::= & \text{import } m & \text{(module import)} \\
 & \mid & \text{def } f(x^{\mathrm{p}}_1, \ldots, x^{\mathrm{p}}_n) : p^{\mathrm{p}} & \text{(local function definition)} \\
 & \mid & x^{\mathrm{p}} = e^{\mathrm{p}} & \text{(assignment)} \\
e^{\mathrm{p}} & ::= & x^{\mathrm{p}} & \text{(variable)} \\
 & \mid & m & \text{(module)} \\
 & \mid & (e^{\mathrm{p}}_1, \ldots, e^{\mathrm{p}}_n) & \text{(tuple)} \\
 & \mid & [e^{\mathrm{p}}_1, \ldots, e^{\mathrm{p}}_n] & \text{(list)} \\
 & \mid & f(e^{\mathrm{p}}_1, \ldots, e^{\mathrm{p}}_n) & \text{(local function application)} \\
 & \mid & m.f^{\mathrm{p}}(e^{\mathrm{p}}_1, \ldots, e^{\mathrm{p}}_n) & \text{(foreign module function application)} \\
 & \mid & x^{\mathrm{p}}.f^{\mathrm{p}}(e^{\mathrm{p}}_1, \ldots, e^{\mathrm{p}}_n) & \text{(foreign class method application)} \\
p^{\mathrm{c}} & ::= & [s^{\mathrm{c}}_1, ..., s^{\mathrm{c}}_n] & \text{(C program)} \\
s^{\mathrm{c}} & ::= & cType\ x^{\mathrm{c}} & \text{(variable type declaration)} \\
 & \mid & x^{\mathrm{c}} = e^{\mathrm{c}} & \text{(assignment)} \\
 & \mid & \text{PyMethodDef } \{f^{\mathrm{p}}, f^{\mathrm{c}}, flag, \ldots\} & \text{(foreign function declaration)} \\
 & \mid & \text{PyObject* } f^{\mathrm{c}}(\text{PyObject* } self, \text{ PyObject* } args)\ \{p^{\mathrm{c}}\} & \text{(foreign function definition)} \\
 & \mid & \text{return } e^{\mathrm{c}} & \text{(function return)} \\
e^{\mathrm{c}} & ::= & x^{\mathrm{c}} & \text{(variable)} \\
 & \mid & \iota_{\mathrm{ap}}\ (\text{args}, u_1 \ldots u_n, x^{\mathrm{c}}_1, \ldots, x^{\mathrm{c}}_n) & \text{(argument parsing)} \\
 & \mid & \iota_{\mathrm{vb}}\ (u_1 \ldots u_m, x^{\mathrm{c}}_1, \ldots, x^{\mathrm{c}}_m) & \text{(value building)} \\
 & \mid & \iota_{\mathrm{ec}} & \text{(explicit conversion)}
\end{array}
$$

where *flag* $\in \mathbb{F}$, $\iota_{\mathrm{ap}} \in \mathbb{I}_{\mathrm{ap}}$, $\iota_{\mathrm{vb}} \in \mathbb{I}_{\mathrm{vb}}$, $\iota_{\mathrm{ec}} \in \mathbb{I}_{\mathrm{ec}}$. $\mathbb{F}$ is a set of bit field flags with the naming prefix `METH_` for calling conventions defined in Python/C API, $\mathbb{I}_{\mathrm{ap}}$, $\mathbb{I}_{\mathrm{vb}}$ and $\mathbb{I}_{\mathrm{ec}}$ are Python/C API families for argument parsing, value building, and explicit conversion respectively.

Fig. 3. Abstract syntax of Python/C multilingual system.

instantiated with a type set, and it is also a type outside the CPython runtime. An argument of a foreign function can be of union type, which is a common feature in the C language. Types like `module` and `iterator` are omitted because they cannot be represented with features in C and will be treated as `object` when passed through foreign function interface.

$$
\begin{array}{lcl}
\tau, \eta & ::= & \texttt{pUnicode} \mid \texttt{pBytearray} \\
 & \mid & \texttt{pBytes} \mid \texttt{pArray} \mid \texttt{pMemoryview} \\
 & \mid & \texttt{pInt} \mid \texttt{pFloat} \mid \texttt{pComplex} \mid \texttt{pInt\_nonnegative} \\
 & \mid & \texttt{pObject} \mid \texttt{pBool} \mid \texttt{pNone} \\
 & \mid & pProduct(\tau_1, \ldots, \tau_n) \mid pUnion(\tau_1, \ldots, \tau_n) \\
 & \mid & pFunc([\tau_1, \ldots, \tau_n], \tau_r)
\end{array}
$$

Fig. 4. Types on the Python side ($pType$: $\tau, \eta \in \mathbb{T}_{\mathrm{p}}$).

At the same time, call interface descriptions can actually indicate not only type conversions, but also stricter value constraints than built-in types. For example, format units (see Section IV-D2 for details) "`b, B, H, I, k, K`" require that the argument passed to a foreign function call must be a nonnegative integer. So we introduce `pInt_nonnegative` as a subtype of `pInt`. Given a subtyping relation $<:$, τ' is a subtype of τ ($\tau' <: \tau$) if any term of type τ' can be used in the context where a term of type τ is expected. Fig. 5 lists subtyping rules of types on the Python side.

Functions are contravariant in their argument types and covariant in their return types as usual. `pBool` indicates an object that accepts any valid Python value that can be tested for

$$\frac{}{\tau <: \tau}$$

$$\frac{}{\tau <: \texttt{pObject}}$$

$$\frac{}{\texttt{pInt\_nonnegative} <: \texttt{pInt}}$$

$$\frac{}{\texttt{pInt} <: \texttt{pFloat}}$$

$$\frac{\tau_i <: \eta_i \quad 1 \le i \le n}{pProduct(\tau_1, \ldots, \tau_n) <: pProduct(\eta_1, \ldots, \eta_n)}$$

$$\frac{set(\tau_1, \ldots, \tau_n) \subseteq set(\eta_1, \ldots, \eta_n)}{pUnion(\tau_1, \ldots, \tau_n) <: pUnion(\eta_1, \ldots, \eta_n)}$$

$$\frac{\tau_i <: \eta_i \quad 1 \le i \le n \quad \eta_r <: \tau_r}{pFunc([\eta_1, \ldots, \eta_n], \eta_r) <: pFunc([\tau_1, \ldots, \tau_n], \tau_r)}$$

Fig. 5. Subtyping rules of types on the Python side.

truth. In Python, a bool object can be any type with attribute `__bool__` or `__len__`. Original type will distract our type conversion rules, so we treat `pBool` as a subtype of `pObject`, rather than a subtype of `pInt` in Python's type system as a single language. Subtype relation between `pInt` and `pFloat` is also outside the Python runtime, as an integer can be cast implicitly when passed to a foreign function.

$$
\begin{array}{lcl}
\alpha, \beta & ::= & \alpha * \mid \texttt{const } \alpha \mid \texttt{unsigned } \alpha \mid \\
 & \mid & \texttt{char} \mid \texttt{short int} \mid \texttt{int} \mid \texttt{long} \mid \texttt{long long} \\
 & \mid & \texttt{float} \mid \texttt{double} \mid \texttt{void} \\
 & \mid & \texttt{Py\_UNICODE} \mid \texttt{PyByteArrayObject} \\
 & \mid & \texttt{PyBytesObject} \mid \texttt{Py\_buffer} \\
 & \mid & \texttt{Py\_ssize\_t} \mid \texttt{Py\_complex} \mid \texttt{PyObject} \\
 & \mid & cProduct(\alpha_1, \ldots, \alpha_n) \mid cUnion(\alpha_1, \ldots, \alpha_n) \\
 & \mid & cFunc([\alpha_1, \ldots, \alpha_n], \alpha_r)
\end{array}
$$

Fig. 6. Types on the C side ($cType$: $\alpha, \beta \in \mathbb{T}_{\mathrm{c}}$).

Types on the C side include common C types like `int`, C structures used to describe built-in types in CPython like `Py_UNICODE`, and those introduced to support language features in type system like *cProduct*. As types on the Python side are all prefixed with 'p', we use the common C type notation as usual without causing ambiguity.

IV. Type System

A multilingual type system consists of two type sets from a host language and a foreign language, as well as a set of rules for assigning types to program constructs in the given context on different language sides. We have defined type sets and program constructs before, and in this section we focus on type inference rules.

A. Hypothetical Judgments

We first define some hypothetical judgments used in our type system.

The hypothetical judgment of the form

$$\Gamma \vdash e :: \tau \qquad \text{(Type assignment)}$$

states that construct e has type τ under the typing context Γ, where Γ consists of hypotheses of the form $x :: \tau$, one for each variable x on host or foreign sides (marked with superscript p and c like before). Since Python uses single colon in its syntax, we use double colons for type assignment instead.

Local functions are functions whose definition and application are both on the host side; while *foreign functions* are those whose application is on the host side and definition is on the foreign side. Foreign function declaration of the form

$$\Gamma^{\mathrm{c}} \vdash f^{\mathrm{p}} \xhookrightarrow{flag} f^{\mathrm{c}} \qquad \text{(Foreign function declaration)} \qquad \text{(D)}$$

maps a foreign function name f^{p} called on the host side to its function definition f^{c} implemented on the foreign side. As a dynamically typed language, foreign function of Python is declared with a calling convention flag instead of explicit type annotations. This bit field specifies how the foreign function call should be constructed, including number of positional or keyword arguments it takes.

$$\Gamma^{\mathrm{c}} \vdash pType\ {}_{\mathrm{p}}\overset{\mathbb{P}}{\mapsto}_{\mathrm{c}}\ cType \qquad \text{(Parameter type conversion)} \qquad \text{(P)}$$

$$\Gamma^{\mathrm{c}} \vdash cType\ {}_{\mathrm{c}}\overset{\mathbb{P}}{\mapsto}_{\mathrm{p}}\ pType \qquad \text{(Return type conversion)} \qquad \text{(R)}$$

By introducing property $\mathbb{P}$ representing different forms of type conversion agreements, we can reason the type constraints imposed on variables on both language sides and conversion rules between them. Program properties include format units of Python/C API families for argument parsing and value building, semantics of certain Python/C APIs, and some static analyses reasoning on interface code units in different intermediate representations like AST (abstract syntax tree) and CFG (control flow graph).

Some judgments for type conversion are feasible only if a certain predicate of program property $\mathbb{P}$ is true. We formulate the relation with the following judgment:

$$\{\pi(\mathbb{P})\}?\{J\}$$

where $\pi \in \Pi$, Π is a set of gated semantic predicates similar to the concept mentioned in [17], and J is a judgment in the form of (P) in our system.

B. Type Inference

Foreign function declaration (D), parameter type conversion (P), and return type conversion (R) together constitute the premises of a type inference rule, which decide the type signature of a foreign function.

$$\frac{D \quad P \quad R}{f^{\mathrm{p}} :: pFunc(\circ, \diamond)} \qquad \text{(Type inference)} \qquad \text{(TInfer)}$$

where parameter type $\circ$ and return type $\diamond$ will be decided by a combination of specific premises.

In the following subsections, we will formulate different forms of these three kinds of premises.

C. Foreign Function Declaration

This part of inference premise builds the relation between foreign function name and its C implementation, denoted as judgment (D). Common calling convention given by $flag$ contains bit field values such as `METH_VARARGS`, `METH_NOARGS`, `METH_O`, etc. The typical `METH_VARARGS` indicates that the foreign function takes one or more Python arguments as input and packs them into one parameter on the C side, which is the second parameter (often called `args`) of C implementation. `METH_NOARGS` is applied to a foreign function with no parameters, passing argument to such foreign function will get a `TypeError` at runtime. `METH_O` is applied to a foreign function with a single object argument. More calling convention flags are embedded in our system according to Python/C API reference manual [23] and CPython implementation.

D. Parameter Type Conversion

We formulate two different forms of parameter type conversion rules.

1) Calling Convention Analysis: As we can see, calling convention flag has effect on the parameter type of a foreign function. With `METH_NOARGS`, it requires the foreign function to be parameter-free, which can be denoted as:

$$\{\mathcal{P}_{\mathrm{PFA}}(flag)\}?\{\Gamma^{\mathrm{c}} \vdash \texttt{pNone}\ {}_{\mathrm{p}}\mapsto_{\mathrm{c}}\ \texttt{void}\} \qquad \text{(PFA)}$$

$\mathcal{P}_{\mathrm{PFA}}$ (Parameter-Free Analysis) is a gated semantic predicate, it analyzes the calling convention flag, and the judgment of parameter type conversion from `pNone` to `void` is viable only when the predicate is true.

However, calling convention flag is not the only determinant for whether a foreign function has no parameters, explicit information in the interface layer can be redundant. The second parameter `args` of f^{c} is to pack all the arguments passed to a foreign function call, if `args` is not used in the function body of f^{c}, then the foreign function is actually parameter-free in implementation. Similarly, we denote this as another gated semantic predicate $\mathcal{P}_{\mathrm{UPA}}$ (Unused Parameter Analysis):

$$\{\mathcal{P}_{\mathrm{UPA}}(f^{\mathrm{c}})\}?\{\Gamma^{\mathrm{c}} \vdash \texttt{pNone}\ {}_{\mathrm{p}}\mapsto_{\mathrm{c}}\ \texttt{void}\} \qquad \text{(UPA)}$$

This kind of behavior should match the calling convention flag, in which case the violation will result in a compile time error. Otherwise, argument of any type can be passed to a foreign function which is designed to be parameter-free. From a type system view, this mismatch will trigger a bug of mixing argument type `pNone` with $pUnion(\texttt{pNone}, \dots)$, which can be any *pType*. $pUnion(\texttt{pNone}, \dots)$ is equivalent to syntax `Optional[Any]` in [5] for ease of understanding.

When predicates $\mathcal{P}_{\mathrm{PFA}}$ and $\mathcal{P}_{\mathrm{UPA}}$ are both true and match with each other, the foreign function is to be parameter-free indeed. This can be easily formulated with a conjunctive normal form of gated semantic predicates.

$$\{\mathcal{P}_{\mathrm{PFA}}(flag) \wedge \mathcal{P}_{\mathrm{UPA}}(f^{\mathrm{c}})\}?\{\Gamma^{\mathrm{c}} \vdash \texttt{pNone}\ {}_{\mathrm{p}}\mapsto_{\mathrm{c}}\ \texttt{void}\}$$
$$\text{(Calling convention)} \qquad \text{(Pcc)}$$

2) Argument Parsing Analysis: When predicate $\mathcal{P}_{\mathrm{PFA}}$ and $\mathcal{P}_{\mathrm{UPA}}$ are both false, the foreign function will take at least one argument, we then describe further judgment form of parameter type conversion based on $\mathbb{I}_{\mathrm{ap}}$, a family of Python/C API for argument parsing.

Python/C API functions for argument parsing use a format string to tell the foreign function about the expected

TABLE I
PARAMETER TYPE CONVERSION IMPOSED BY FORMAT UNITS

Strings and buffers	**Numbers**
pUnicode ${}_{p}\!\xmapsto{\text{s}}_{c}$ const char *	pInt_nonnegative ${}_{p}\!\xRightarrow{\text{b}}_{c}$ unsigned char
pUnicode $\mid$ pByteslike ${}_{p}\!\xmapsto{\text{s*}}_{c}$ Py_buffer	pInt_nonnegative ${}_{p}\!\xmapsto{\text{B}}_{c}$ unsigned char
pUnicode $\mid$ pImmbyteslike ${}_{p}\!\xmapsto{\text{s\#}}_{c}$ (const char *, int $\mid$ Py_ssize_t)	pInt ${}_{p}\!\xRightarrow{\text{h}}_{c}$ short int
pUnicode $\mid$ pNone ${}_{p}\!\xmapsto{\text{z}}_{c}$ const char *	pInt_nonnegative ${}_{p}\!\xmapsto{\text{H}}_{c}$ unsigned short int
pUnicode $\mid$ pByteslike $\mid$ pNone ${}_{p}\!\xmapsto{\text{z*}}_{c}$ Py_buffer	pInt ${}_{p}\!\xRightarrow{\text{i}}_{c}$ int
pUnicode $\mid$ pImmbyteslike $\mid$ pNone ${}_{p}\!\xmapsto{\text{z\#}}_{c}$ (const char *, int $\mid$ Py_ssize_t)	pInt_nonnegative ${}_{p}\!\xmapsto{\text{I}}_{c}$ unsigned int
pImmbyteslike ${}_{p}\!\xmapsto{\text{y}}_{c}$ const char *	pInt ${}_{p}\!\xRightarrow{\text{l}}_{c}$ long int
pByteslike ${}_{p}\!\xmapsto{\text{y*}}_{c}$ Py_buffer	pInt_nonnegative ${}_{p}\!\xmapsto{\text{k}}_{c}$ unsigned long
pImmbyteslike ${}_{p}\!\xmapsto{\text{y\#}}_{c}$ (const char *, int $\mid$ Py_ssize_t)	pInt ${}_{p}\!\xRightarrow{\text{L}}_{c}$ long long
pBytearray ${}_{p}\!\xmapsto{\text{w*}}_{c}$ Py_buffer	pInt_nonnegative ${}_{p}\!\xmapsto{\text{K}}_{c}$ unsigned long long
pUnicode ${}_{p}\!\xmapsto{\text{u}}_{c}$ const Py_UNICODE *	pInt ${}_{p}\!\xRightarrow{\text{n}}_{c}$ Py_ssize_t
pUnicode ${}_{p}\!\xmapsto{\text{u\#}}_{c}$ (const Py_UNICODE *, int $\mid$ Py_ssize_t)	pBytes[1] $\mid$ pBytearray[1] ${}_{p}\!\xRightarrow{\text{c}}_{c}$ char
pUnicode $\mid$ pNone ${}_{p}\!\xmapsto{\text{Z}}_{c}$ const Py_UNICODE *	pUnicode[1] ${}_{p}\!\xRightarrow{\text{C}}_{c}$ int
pUnicode $\mid$ pNone ${}_{p}\!\xmapsto{\text{Z\#}}_{c}$ (const Py_UNICODE *, int $\mid$ Py_ssize_t)	pFloat ${}_{p}\!\xRightarrow{\text{f}}_{c}$ float
pBytes ${}_{p}\!\xmapsto{\text{S}}_{c}$ PyBytesObject *	pFloat ${}_{p}\!\xRightarrow{\text{d}}_{c}$ double
pBytearray ${}_{p}\!\xmapsto{\text{Y}}_{c}$ PyByteArrayObject *	pComplex ${}_{p}\!\xRightarrow{\text{D}}_{c}$ Py_complex
pUnicode ${}_{p}\!\xmapsto{\text{U}}_{c}$ Py_UNICODE *	**Objects**
pUnicode ${}_{p}\!\xmapsto{\text{es}}_{c}$ (const char *, char **)	pObject ${}_{p}\!\xmapsto{\text{O}}_{c}$ PyObject *
pUnicode $\mid$ pBytes $\mid$ pBytearray ${}_{p}\!\xmapsto{\text{et}}_{c}$ (const char *, char **)	pObject ${}_{p}\!\xmapsto{\text{O!}}_{c}$ (α *, PyObject *)
pUnicode ${}_{p}\!\xmapsto{\text{es\#}}_{c}$ (const char *, char **, int $\mid$ Py_ssize_t)	pObject ${}_{p}\!\xmapsto{\text{O\&}}_{c}$ ((PyObject *, void *) $\rightarrow$ status, any *)
pUnicode $\mid$ pBytes $\mid$ pBytearray ${}_{p}\!\xmapsto{\text{et\#}}_{c}$ (const char *, char **, int $\mid$ Py_ssize_t)	pBool ${}_{p}\!\xmapsto{\text{p}}_{c}$ status
Special characters: $\mid$ \$: ;	$(\tau_1, \ldots)$ ${}_{p}\!\xmapsto{\text{(format unit 1, ...)}}_{c}$ $(\alpha_1, \ldots)$

${}_{p}\!\xRightarrow{\textit{format unit}}_{c}$ *describes a conversion for numbers with extra overflow checking at runtime.*

arguments, their number, types, and value constraints. Some constraints are not bound to CPython built-in types, but informal specifications written in the reference manual, and are easily overlooked to trigger runtime crash or unexpected behaviors. We use refined subtypes for some common value constraints like non-negative integers (see Section III-C).

A format string consists of zero or more format units. A format unit describes one Python object, which consists of one or more characters, or a parenthesized sequence of format units. For example, `PyArg_ParseTuple` in Fig. 1(c) is a typical Python/C API function in $\mathbb{I}_{\text{ap}}$, its format string `"II"` consisting of two format units '`I`' indicates that the foreign function `foo` takes two Python integer objects which will be parsed as two C variables of type `unsigned int`.

For an argument parsing function $\iota_{\text{ap}} \in \mathbb{I}_{\text{ap}}$, format string of ι_{ap} is a sequence of format units "$u_1, \ldots, u_n$", each of which indicates a type conversion rule for one Python argument passed to the foreign function:

$$\Gamma^{c} \vdash \tau_i \;{}_{p}\!\xmapsto{\iota_{\text{ap}}.u_i}_{c}\; \alpha_i \quad 1 \leq i \leq n \quad \text{(Argument parsing)} \quad \text{(Pap)}$$

TABLE I shows the complete parameter type conversion rules imposed by format units. To save space, on both language sides, union type $union(\tau_1, \tau_2)$ is denoted as $\tau_1 \mid \tau_2$, product type $product(\tau_1, \tau_2)$ is denoted as (τ_1, τ_2), and function type $func(\tau_{in}, \tau_r)$ is denoted as $\tau_{in} \rightarrow \tau_r$. Python/C interface code is C code using the Python/C API, which in CPython are declared in `Python.h` and other transitively included C header files. Typing context of the analyses based on Python/C API is always Γ^{c}, so it is omitted in the table as well.

Strings and buffers are formats units that allow an object to be accessed as a contiguous chunk of memory. The typical format unit "s" takes a Python Unicode object as positional argument, and converts it to a C pointer to a character string. This format does not accept bytes-like objects (*e.g.,* file system paths). `pImmbyteslike` is a union of immutable bytes-like objects `pBytes`, `pArray` and `pMemoryview`. `pByteslike` includes `pImmbyteslike` and the mutable byte array `pBytearray`.

Some conversion result (format units marked with #) of a Python argument is stored into another C variable of type `int` or `Py_ssize_t`, representing the length of the argument object. When a format sets a pointer to a buffer, the buffer is managed by the corresponding Python object, and the buffer shares the lifetime of this object. Programmers do not have to release any memory by themselves. The only exceptions are "es", "es#", "et", and "et#", which take another C variable of type `char **` when converting the argument with a specified encoding like 'utf-8'. When a `Py_buffer` structure gets filled (format units marked with *, *e.g.,* "s*"), the underlying buffer is locked so that the caller can subsequently use the buffer even inside a multithread block without worry about mutable data being resized or destroyed.

As shown in right column of TABLE I, format units for numbers using ${}_{\mathrm{p}}\overset{\text{format unit}}{\Longmapsto}{}_{\mathrm{c}}$ convert the argument with additional overflow checking at runtime. Python has a feature supporting long integers instead of different sizes of types. With subtypes using type refinement [24], we can bring the overflow checking to compile time. A refinement type is a type endowed with a predicate which is assumed to hold for any element of the refined type. Here we take `pInt_nonnegative` as an example, Python integer can be divided into more intervals reflecting different sizes of C types. `Py_ssize_t` has the same length as the compiler's `size_t`, but is signed. Thus using `Py_ssize_t` instead of C `int` can avoid only indexing 2^{31} elements of a sequence on a 64-bit machines [25]. We use $pType[n]$ to represent an additional length constraint on a $pType$, note to distinguish it from an array. For example, format unit "C" converts a Python character, represented as a Unicode object of length 1, to a C `int`.

Below the right column of TABLE I, format units for objects store the Python object in the C object pointer. Format unit "O" does not attempt any conversion like strings and buffers format units "S/Y/U". The foreign C program directly receives the argument object that was passed from the Python side. "O!" takes an another C variable which is the address of a Python type object in C structure, the Python object passed to the foreign function should have the required type, otherwise a `TypeError` will be raised. "O&" converts a Python object to a C variable through a converter function. The converted C variable can be of arbitrary type according to the converter function, in which the result is stored in a address of type `void *`. The returned `status` should be an integer of value 1 for a successful conversion and 0 if the conversion has failed. Format unit "p" tests the value passed in for truth (a boolean predicate) and converts the result to its equivalent C integer value of 1/0. When a Python sequence of product type is passed to a foreign function, the conversion will follow a parenthesized sequence of format units. Format units for sequences may be nested as well.

Some other characters have special meanings in the format string. "|" indicates that the remaining arguments in the Python argument list are optional. "$" indicates that the remaining arguments in the Python argument list are keyword-only. ":" and ";" indicate the ending of the format units list, the string after them is used in error message.

Distinguished from other text based type inference methods [7], [26], our format string parser based on stricter parameter type conversion rules is sound and deterministic.

E. Return Type Conversion

We formulate four different forms of return type conversion rules.

1) Value Building Analysis: Python has a language feature called multiple returns, to support this for a foreign function implemented in C, a family of Python/C API functions for value building ($\mathbb{I}_{\mathrm{vb}}$) work similar to those for argument parsing ($\mathbb{I}_{\mathrm{ap}}$). Python/C APIs for value building create a new value based on a format string, following the rules in TABLE I in reverse. For example, `PyLong_FromLong(x * y)` in line 5 of Fig. 1(c) is equivalent to `Py_BuildValue("I", x * y)`, where format unit '`I`' converts a C unsigned int to a Python integer object. Value building can be denoted as:

$$\Gamma^{\mathrm{c}} \vdash \beta_j \, {}_{\mathrm{c}}\!\xmapsto{\iota_{\mathrm{vb}}.u_j}{}_{\mathrm{p}}\, \eta_j \quad 1 \leq j \leq m, \eta = (\eta_1, \dots, \eta_m) \qquad \text{(Value building)} \quad \text{(Rvb)}$$

Multiple C variables of type $\beta_1, \dots, \beta_j$ are converted to Python objects of type $\eta_1, \dots, \eta_j$, respectively, tuple η will be the return of the foreign function with multiple returns.

2) Explicit Conversion Analysis: When a foreign function returns only one Python object which is not a tuple, Python/C APIs for explicit conversion can directly indicate a return type conversion rule from a C variable to a Python object. It can be denoted as:

$$\Gamma^{\mathrm{c}} \vdash \beta \, {}_{\mathrm{c}}\!\xmapsto{\iota_{\mathrm{ec}}}{}_{\mathrm{p}}\, \eta \qquad \text{(Explicit conversion)} \qquad \text{(Rec)}$$

Explicit return type conversion Python/C API set $\mathbb{I}_{\mathrm{ec}}$ contains APIs in the form of: (i) `PyPT_FromCT()` (*e.g.,* `PyLong_FromLong`) which converts a C variable of $cType$ `CT` to Python object of $pType$ `PT`, (ii) `PyPT_New()` (*e.g.,* `PyList_New`) which returns a new Python object of $pType$ `PT`, and (iii) `Py_PT` (*e.g.,* Py_None) is an object of $pType$ `PT` itself.

3) Type Cast Analysis: C program supports explicit type cast as a right-associative operator, for a return statement in the form of `return` $(\beta_1)\dots(\beta_n)e$, we can infer that:

$$\Gamma^{\mathrm{c}} \vdash (\beta_1)\dots(\beta_n)e :: \beta_1 \quad \beta_1 \, {}_{\mathrm{c}}\!\mapsto_{\mathrm{p}} \eta_1 \qquad \text{(Type cast)} \quad \text{(Rtc)}$$

As a return of the C implementation of a Python foreign function, $cType$ β_1 must be a C structure of the objects used to describe Python built-in types. For example, `Py_UNICODE` ${}_{\mathrm{c}}\!\mapsto_{\mathrm{p}}$ `pUnicode` is an intrinsic bijection if no other conversion is imposed on.

4) Reaching Definition Analysis: Consider a more complicated situation in Fig. 7, variable `result` declared as type `T1` will be returned as `PyObject *` in most cases. It can be assigned to another variable or to the value of refined types by calling some Python/C APIs talked before. We analyze the type propagation with an intra-procedural reaching definition analysis, denoted as $\mathcal{T}_{\mathrm{RDA}}$, in which three forms of return type conversion rules before can be called. For example in Fig. 7, type of `result` can be `T1` or `T2` or actual type of variable `tmp`, type declaration (`T1`) can be easily extracted, `T2` can be analyzed through a value building analysis, and type of `tmp` can be analyzed by calling $\mathcal{T}_{\mathrm{RDA}}$ recursively, which will call a explicit conversion analysis in the end. The type list `[T1, T2, T3]` can only be unified when a subtyping rule holds. When inferring using $\mathcal{T}_{\mathrm{RDA}}$, the rule is denoted as:

$$\Gamma^{\mathrm{c}} \vdash e :: \mathcal{T}_{\mathrm{RDA}}(e) \qquad \text{(Reaching definition)} \qquad \text{(Rrd)}$$

```
T1 result, tmp;
result = Py_BuildValue(...); // T2
if (...) {
  tmp = PyT3_New(...);
  result = tmp;
}
return result;
```

Fig. 7. An example for complicated return type conversion.

where $\mathcal{T}_{\text{RDA}}$ takes the variable name to be returned as input.

Conclusion: For three premise parts of rule (TInfer), D has the only form (D), P has two forms (Pcc) and (Pap), while R has four forms (Rvb), (Rec), (Rtc) and (Rrd).

For example, the following rule is a typical pattern of foreign function that indicates parameter type conversion with argument parsing analysis and return type conversion with value building analysis.

$$\frac{\begin{array}{c}\Gamma^{\text{c}} \vdash f^{\text{p}} \xhookrightarrow{flag} f^{\text{c}} \\ \Gamma^{\text{c}} \vdash \tau_i \; {}_{\text{p}}\!\xmapsto{\iota_{\text{ap}}.u_i}_{\text{c}} \alpha_i \quad 1 \leq i \leq n \\ \Gamma^{\text{c}} \vdash \beta_j \; {}_{\text{c}}\!\xmapsto{\iota_{\text{vb}}.u_j}_{\text{p}} \eta_j \quad 1 \leq j \leq m \quad \eta = (\eta_1, \ldots, \eta_m)\end{array}}{f^{\text{p}} :: pFunc([\tau_1, \ldots, \tau_n], (\eta_1, \ldots, \eta_m))} \quad \text{(TInfer: D-Pap-Rvb)}$$

Another typical one is the parameter-free foreign function returning with explicit conversion analysis.

$$\frac{\begin{array}{c}\Gamma^{\text{c}} \vdash f^{\text{p}} \xhookrightarrow{flag} f^{\text{c}} \quad \Gamma^{\text{c}} \vdash \beta \; {}_{\text{c}}\!\xmapsto{\iota_{\text{ec}}}_{\text{p}} \eta \\ \{\mathcal{P}_{\text{PFA}}(flag) \land \mathcal{P}_{\text{UPA}}(f^{\text{c}})\}?\{\Gamma^{\text{c}} \vdash \texttt{pNone} \; {}_{\text{p}}\!\mapsto_{\text{c}} \texttt{void}\}\end{array}}{f^{\text{p}} :: pFunc(\texttt{pNone}, \eta)} \quad \text{(TInfer: D-Pcc-Rec)}$$

All the 12 variants compose our type inference rules.

V. Evaluation

In this section, we describe the system architecture of PyCType, demonstrate its effectiveness evaluation, and analyze the mismatch between the foreign function declaration and its implementation.

A. Implementation

We implement our static type inference prototype system for foreign functions of Python in PyCType, and Fig. 8 shows the overall structure of our system. *Language separator* separates Python and C code. *Preprocessor configurator* configures the header files needed to parse the sources, including headers for components of the project, for Python/C API, as well as for system and third-party libraries. *AST parser* is based on a complete C99 parser in Python called *pycparser* [27]. After generating the AST of the interface code, some analysis tasks will be built on *AST visitor*. When an analysis requires other representations such as CFG, *AST transformer* takes the sub-node of the whole AST as input, which can improve time efficiency. *Foreign function declaration* and its *definition* (see abstract syntax in Fig. 3) are common AST sub-nodes used in our system. D-Analyzer extracts premise (D) from *foreign function declaration*, its $flag$ is used to analyze premise (Pcc), and its f^{c} is used for visiting *foreign function definition*, based on which other analyses are constructed. Analyzers Pap, Rvb and Rec are Python/C API based, while analyzers Rtc and Rrd are based on the `return` statement.

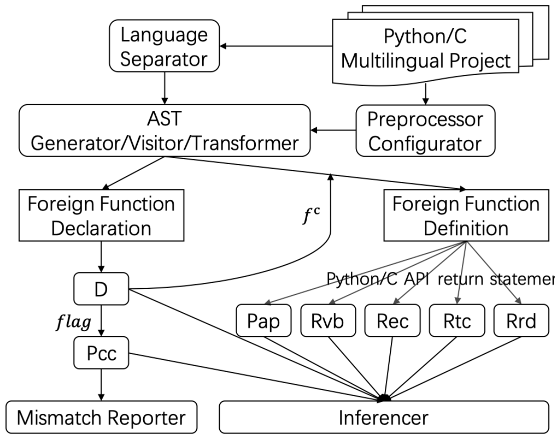

Language Separator
Python/C Multilingual Project
AST Generator/Visitor/Transformer
Preprocessor Configurator
Foreign Function Declaration
f^c
Foreign Function Definition
D
Python/C API
return statement
$flag$
Pap
Rvb
Rec
Rtc
Rrd
Pcc
Mismatch Reporter
Inferencer

Fig. 8. Overview structure of PyCType.

B. Experiments

We conducted two kinds of experiments for demonstrating completeness and effectiveness of PyCType.

1) Completeness: Our type inference is strictly rule-directed with conservative must static analyses, making it sound in design. To evaluate its completeness, we conducted experiments on three representative and widely used Python/C projects, *i.e.,* CPython, NumPy and Pillow. CPython [28] is the default reference implementation of Python. Its standard library contains a large amount of foreign functions. NumPy [29] is the fundamental package for scientific computing with Python. It is the de facto standard for storing multi-dimensional data in scientific Python. Pillow [30] is the de facto image processing standard library in Python, which provides extensive file format support, an efficient internal representation, and fairly powerful image processing capabilities. The computing power of these performance-sensitive libraries comes from underlying C extension modules.

TABLE II
Completeness of Parameter Type Inference

Project	KLOC	Time (s)	Foreign Function	Pcc	Pap	Bug	Coverage
CPython	279.2	159.7	1529	503	606	32	74.6%
NumPy	507.1	76.4	90	9	51	6	73.3%
Pillow	29.0	40.7	132	2	119	10	99.2%
Total	815.3	276.8	1751	514	776	48	76.4%

TABLE II shows the completeness of inferring parameter number and type for foreign functions, parameter type contains more variables than return (single return in most cases) and can be used for type checking directly. Column `KLOC` lists kilo lines of C code in the project. Time elapse for the whole analysis is proportional to the number of foreign functions. On average, PyCType infers type signature of a foreign function in about 158ms. For each foreign function, we set up two rules to infer its parameter types. Column `Coverage` counts how many foreign functions can be inferred by the two rules or reported with a bug, from 73.3% to 99.2%, which is related to the implementation detail of a project. What is not covered by the `Coverage` column are some cases that cannot be inferred at present, accounting for less than 24% on average. For example, `PyPT_AsCT` is a Python/C API family that works similar to `PyPT_FromCT` (see Section IV-E2) but in reverse. It converts a Python object unpacked from `args` parameter to a C variable. It can be regarded as a Python/C API for argument parsing with only one simple (not parenthesized) format unit. The completeness of our inference system can be improved by supporting more Python/C APIs like `PyPT_AsCT` family.

2) Effectiveness: Manually checking the type inference results of Pillow proves that our method is sound as designed, meaning that all typing results are correct but may be imprecise, like inferring with type `pObject`. However, some degree of incompleteness and imprecision is acceptable, because the untyped return object of a foreign function can be an argument of other foreign or local functions, parameter type of which can be inferred or already known. For this reason, the evaluation showing the effectiveness of our system is set up as a type inference enhancement experiment. The state-of-the-art static Python type inference tool Pytype [11] from Google does not support foreign functions, it uses the Typeshed [18] as embedded external type annotations (see Section II-B). By coding our type inference results in Type Hints syntax [5] used by Typeshed, and adding them to Pytype as a complement, we show the effectiveness of PyCType.

We choose Pillow as our experimental object because of its high type inference coverage (see TABLE II). Pillow has 132 foreign functions, 121 of which can be inferred with their type signatures, 63 of which are methods of class `ImagingCore`, and 46 of which are functions of module `Image`. We code these 109 foreign functions in Type Hints. By using GitHub API, we set up an experimental corpus of Python programs using Pillow from projects with more than 30 thousand stars, they cover different application domains like image processing, machine learning, Web framework, etc. TABLE III shows our type inference enhancement results. Some test files have been rewritten to: (1) remove type annotations, (2) remove auxiliary functions without using Pillow, (3) spread nested function calls like `a = f1(f2())` to `b = f2();a = f1(b)`, and (4) assign a variable to every function call (`pNone` is also a type). Column `Pytype` records the inference rate of variables (recall precision) in test files with unmodified Pytype. Column `Pytype+PyCType` is the result after taking type inference results of PyCType as a complement, which shows different degrees of improvement from 7% to 80% (27.5% on average). The enhancement can be further improved, as besides Pillow some files use other packages with foreign functions, which can also be statically inferred with our method.

C. Mismatch Bug

The predicated condition of (Pcc) is a conjunctive normal form of gated semantic predicates $\mathcal{P}_{\text{PFA}}$ from PFA analysis and $\mathcal{P}_{\text{UPA}}$ from UPA analysis. Pcc is viable only if $\mathcal{P}_{\text{PFA}}$ and $\mathcal{P}_{\text{UPA}}$ match each other and are both true. If they match each other and are both false, it indicates that the foreign function is to take at least one parameter, PyCType continues to infer with other parameter type conversion rules like (Pap). However, in some cases that $\mathcal{P}_{\text{PFA}}$ and $\mathcal{P}_{\text{UPA}}$ do not match each other, the redundant but inconsistent explicit information will suspend the type inference and report a bug. By manually checking all the warning reports, there is no false positives, which also proves the soundness of our system. In these cases, a false $\mathcal{P}_{\text{PFA}}$ on the calling convention flag indicates that the foreign function receives arguments of some types, while a true $\mathcal{P}_{\text{UPA}}$ means the opposite—nothing should be passed as it will not be used inside the foreign function. In the single language scenario, it is the traditional `-Wunused-parameter` warning of compilers. To describe in type system, the parameter type of the foreign function is `pNone` in foreign function definition, but actually the argument of any type is legal as foreign function declaration indicates, *i.e.,* type $pUnion(\texttt{pNone}, \ldots)$ (or `Optional[Any]` in type hints). Among all the 48 mismatch bugs we find in CPython, NumPy and Pillow, in 5 cases the unused `args` parameter is marked with compiler attribute `__unused__`, which can avoid potential security attacks. 8 of the reported bugs have been confirmed and fixed by NumPy[2] and Pillow[3] communities till now.

VI. Related Work

A. Python Static Type Inference

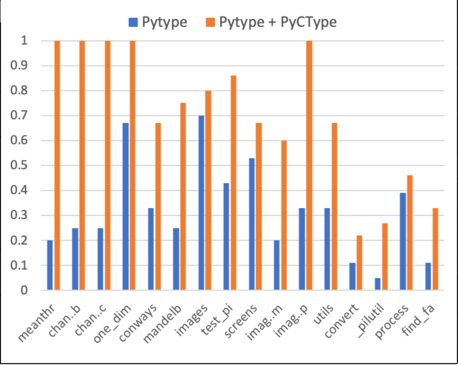
Many existing studies focus on static type inference of Python, which can be divided into three categories.

1) Type Annotation Based: By rewriting the type-sensitive part of Python programs with type hints [5], [31], tools like Mypy [2], Pyright [3] and Pyre [4] can check type misuses based on annotations. These methods do not distinguish between functions defined in local Python and foreign C. However, they cannot be used on a large scale because annotating the source code requires extra workload of experienced engineers and violates the agile coding practice. A suitable scenario for using annotation based type inference is when exposing interface of Python library to external developers.

2) Machine Learning Based: Machine learning methods [6], [7] first tag Python variables, then infer based on the naming rules that have been learned. These methods infer based on variable name text which includes parameters of functions, thus they can support foreign functions as well. At the same time, they get better precision and time efficiency in their tests. However, the precision of machine learning needs to be manually verified, and it is a top-N result, which means every type they infer is correct with a certain probability, which is unacceptable for deterministic tasks.

[2]NumPy-issue-18665: https://github.com/numpy/numpy/issues/18665
[3]Pillow-issue-5487: https://github.com/python-pillow/Pillow/issues/5487

TABLE III
TYPE INFERENCE ENHANCEMENT EXPERIMENT

File	Project	Star	Domain	Pytype	Pytype + PyCType	Improvement
meanthreshold.py	TheAlgorithms/Python	105k	image algorithm	20%	100%	
change_brightness.py				25%	100%	
change_contrast.py				25%	100%	
one_dimensional.py				67%	100%	
conways_game_of_life.py				33%	67%	
mandelbrot.py				25%	75%	
images.py	django/django	57.2k	web framework	70%	80%	
test_pipeline_images.py	scrapy/scrapy	40.5k	web crawling	43%	86%	
screenshots.py	apache/superset	38.3k	data visualization	53%	67%	
image_mobject.py	3b1b/manim	33.4k	math animation engine	20%	60%	
image_processing.py	home-assistant/core	42.4k	home automation	33%	100%	
utils.py	tensorflow/models	69.8k	machine learning	33%	67%	
convert_to_tflite.py				11%	22%	
_pilutil.py	scikit-learn/scikit-learn	45.6k		5%	27%	
processing_image.py	huggingface/transformers	45.5k		39%	46%	
find_faces_in_picture.py	ageitgey/face_recognition	39.8k		11%	33%	

3) Static Program Analysis: Since type annotation and machine learning based methods are rarely applied in industrial practice, more studies infer with static program analysis techniques like data flow analysis [10], abstract interpretation [8], [11], [12], satisfiability modulo theories (SMT) [9], etc. These methods produce deterministic results and require no modification of source. On the other hand, they can provide automatic type annotations and tags for two categories of studies before. However, all these studies do not consider foreign functions, making type recall of variables stay at a medium level and not suitable for multilingual software architecture commonly used today.

B. Multilingual Program Analysis

FFI bridges the gap of different language features between host and foreign languages, which is also the research focus of multilingual program analysis.

Li *et al.* [21], [22] study the problems related to exception handling in the JNI—FFI between Java and C. The exceptions thrown by JNI in the C foreign code do not immediately return control to the JVM. Without additional explicit handling, this can lead to serious safety problems.

Python uses reference counting to manage its heap objects, but the extension modules written in C/C++ are outside the memory management system. So it is necessary to manually manage the reference counts of native objects, which is error-prone because of borrowing/stealing reference [32]. Pungi [19] uses affine abstraction to statically analyze the SSA form of programs, and RID [20] uses inconsistent path pair checking to relax Pungi's hypothesis and improves analysis accuracy.

Empirical studies give a taxonomy of FFI-related bug patterns for JNI [33] and Python/C API [1], which are different from each other because of feature discrepancies between Java and Python. While both of them take type misuse as one of safety concerns, especially for a dynamically typed language.

For type misuses between host and foreign language, Furr *et al.* propose a type inference system for type checking the OCaml/C interface, which can avoid type and memory security problems in the foreign methods [15]. They further extend it to support polymorphic type inference of the JNI [16]. However, OCaml and Java are both host languages using static type checking, which is quite different from dynamically typed languages like Python.

Brown *et al.* [34] describe crash-, type-, and memory-safety errors that JavaScript binding code creates, which is another commonly used dynamically typed language. Its lightweight static bug checker for type errors build an intra-procedural forward code traversal to check if type requirements of target C API is satisfied. It is not a type inference based approach, and this paper agrees that type inference will be helpful but difficult for interface safety of dynamically typed languages like JavaScript and Python.

VII. Conclusion

We propose PyCType, a static type inference system for foreign functions of Python. The type inference rules in PyCType are formulated with three parts of composable premises, which model and analyze implicit information for type conversion in the Python/C interface layer. We demonstrate PyCType on three representative multilingual software and infer type signatures soundly for most foreign functions. As a complement to static type inference tool Pytype, PyCType enables it to analyze Python programs with foreign function calls. We find mismatch bugs that make a parameter-free foreign function take argument of any type, and some bugs we reported have been confirmed and fixed by communities.

Acknowledgment

This work was partially funded by the National Natural Science Foundation of China (No. 61772487), Anhui Provincial Natural Science Foundation (No. 1808085MF198), and Open Research Fund Program of Beijing National Research Center for Information Science and Technology.

References

[1] M. Hu and Y. Zhang, “The Python/C API: Evolution, usage statistics, and bug patterns,” in *27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 532–536.

[2] J. Lehtosalo, “Mypy: Optional static typing for Python.” [Online]. Available: http://mypy-lang.org

[3] Microsoft, “Pyright: Static type checker for Python.” [Online]. Available: https://github.com/microsoft/pyright

[4] Facebook, “Pyre: Performant type checker for Python.” [Online]. Available: https://github.com/facebook/pyre-check

[5] G. van Rossum, J. Lehtosalo, and Łukasz Langa, “PEP 484: Type hints,” 2014. [Online]. Available: https://www.python.org/dev/peps/pep-0484/

[6] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, “Typilus: Neural type hints,” in *41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020, pp. 91–105.

[7] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, “Python probabilistic type inference with natural language support,” in *24th ACM SIGSOFT Symposium on the Foundation of Software Engineering (FSE)*, 2016, pp. 607–618.

[8] R. Monat, A. Ouadjaout, and A. Miné, “Static type analysis by abstract interpretation of Python programs,” in *European Conference on Object-Oriented Programming*, 2020.

[9] M. Hassan, C. Urban, M. Eilers, and P. Müller, “Maxsmt-based type inference for Python 3,” in *International Conference on Computer-Aided Verification*, 2018, pp. 12–19.

[10] L. Fritz and J. Hage, “Cost versus precision for approximate typing for Python,” in *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 2017, pp. 89–98.

[11] Google, “Pytype: Static type analyzer for Python code.” [Online]. Available: https://github.com/google/pytype

[12] Y. Wang, “PySonar2: An advanced semantic indexer for Python.” [Online]. Available: https://github.com/yinwang0/pysonar2

[13] Wikipedia, “Foreign function interface.” [Online]. Available: https://en.wikipedia.org/wiki/Foreign_function_interface

[14] A. Green *et al.*, “libffi: A portable foreign function interface library.” [Online]. Available: http://sourceware.org/libffi/

[15] M. Furr and J. S. Foster, “Checking type safety of foreign function calls,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 62–72.

[16] ——, “Polymorphic type inference for the JNI,” in *15th European Conference on Programming Languages and Systems (ESOP)*, 2006, pp. 309–324.

[17] T. Parr, S. Harwell, and K. Fisher, “Adaptive LL(*) parsing: The power of dynamic analysis,” in *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2014, pp. 579–598.

[18] Python, “Typeshed: Collection of library stubs for Python with static types.” [Online]. Available: https://github.com/python/typeshed

[19] S. Li and G. Tan, “Finding reference-counting errors in Python/C programs with affine analysis,” in *European Conference on Object-Oriented Programming*, 2014, pp. 80–104.

[20] J. Mao, Y. Chen, Q. Xiao, and Y. Shi, “RID: Finding reference count bugs with inconsistent path pair checking,” in *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 531–544.

[21] S. Li and G. Tan, “Finding bugs in exceptional situations of JNI programs,” in *16th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2009, pp. 442–452.

[22] ——, “JET: Exception checking in the Java native interface,” in *26th ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011, pp. 345–358.

[23] “Python/C API reference manual.” [Online]. Available: https://docs.python.org/3/c-api/index.html

[24] T. Freeman and F. Pfenning, “Refinement types for ML,” in *ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, 1991, pp. 268–277.

[25] M. von Löwis, “PEP 353: Using ssize_t as the index type,” Dec. 2005. [Online]. Available: https://www.python.org/dev/peps/pep-0353/

[26] R. S. Malik, J. Patra, and M. Pradel, “Nl2type: inferring JavaScript function types from natural language information,” in *41th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019, pp. 304–315.

[27] E. Bendersky, “Pycparser: Complete C99 parser in pure Python.” [Online]. Available: https://github.com/eliben/pycparser

[28] G. Van Rossum *et al.*, “Python programming language.” in *USENIX Annual Technical Conference*, vol. 41, 2007, p. 36.

[29] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[30] A. Clark *et al.*, “Pillow: Python imaging library.” [Online]. Available: https://python-pillow.org/

[31] C. Winter and T. Lownds, “PEP 3107: Function annotations,” 2006. [Online]. Available: https://www.python.org/dev/peps/pep-3107/

[32] A. J. H. Simons, “Borrow, copy or steal?: Loans and larceny in the orthodox canonical form,” in *13th SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1998, pp. 65–83.

[33] G. Tan and J. Croft, “An empirical security study of the native code in the JDK,” in *17th USENIX Conference on Security Symposium (USENIX Security)*, 2008, pp. 365–377.

[34] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan, “Finding and preventing bugs in JavaScript bindings,” in *IEEE Symposium on Security and Privacy*. IEEE, 2017, pp. 559–578.