PoTo: A Hybrid Andersen's Points-to Analysis for Python

Ingkarat Rak-amnouykit
Rensselaer Polytechnic Institute, USA

Ana Milanova
Rensselaer Polytechnic Institute, USA

Guillaume Baudart
Université Paris Cité, Inria, CNRS, IRIF, France

Martin Hirzel
IBM Research, USA

Julian Dolby
IBM Research, USA

Abstract

As Python is increasingly being adopted for large and complex programs, the importance of static analysis for Python (such as type inference) grows. Unfortunately, static analysis for Python remains a challenging task due to its dynamic language features and its abundant external libraries. To help fill this gap, this paper presents PoTo, an Andersen-style context-insensitive and flow-insensitive points-to analysis for Python. PoTo addresses Python-specific challenges and works for large programs via a novel hybrid evaluation, integrating traditional static points-to analysis with concrete evaluation in the Python interpreter for external library calls. We evaluate the analysis with two clients: type inference and call graph construction. This paper presents PoTo+, a static type inference for Python built on PoTo. We evaluate PoTo+ and compare it to two state-of-the-art Python type inference techniques: (1) the static rule-based Pytype and (2) the deep-learning based DLInfer. Our results show that PoTo+ outperforms both Pytype and DLInfer on existing Python packages. This paper also presents PoToCG, a call-graph construction analysis for Python built on PoTo. We compare PoToCG to PyCG, the state of the art for this problem, and show that PoTo produces more complete and more precise call graphs.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Python, Points-to analysis, Machine learning libraries

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.26

Acknowledgements We thank the ECOOP reviewers for their valuable comments and the US National Science Foundation for supporting our work through awards #1814898 and #2232061.

1 Introduction

Points-to analysis is a fundamental static analysis that determines what objects a reference variable may point to. It has a wide variety of applications (aka. *client* analyses), including call graph construction, type inference, debugging, and security vulnerability detection. Essentially all interesting questions one could ask about a program require some form of points-to information and points-to analysis has been studied extensively in the context of mainstream languages such as C, C++, Java, and JavaScript. Surprisingly, points-to analysis for Python has received less attention, despite Python's popularity.

This paper presents PoTo, an Andersen-like point-to analysis for Python. Andersen's analysis [2] is a classical flow-insensitive, context-insensitive, and inclusion-constraint-based analysis. It computes a points-to graph *Pt* where the nodes are reference variables and heap

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva; Article No. 26; pp. 26:1–26:29
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

objects, and the edges represent points-to relations. For example, an edge $x \to o$ represents that x points to heap object o (i.e., reference x stores the address of o), and $o_1 \xrightarrow{f} o_2$ represents that field f of o_1 points to o_2.

While PoTo leverages well-known techniques, it also adapts to the unique challenges of Python: complex syntax and module system, dynamic semantics, and ubiquitous use of external libraries whose code is unavailable. First, our solution presents principled translation from Python source into the 3-address code intermediate representation demanded by points-to analysis. It specifies explicitly which Python features are handled precisely and faithfully to the feature semantics and which are handled approximately via a default interpretation; due to the complexity of Python, handling is unsound. Second, a novelty of PoTo is *hybridization*. We observe that many expressions, particularly ones stemming from external libraries and built-in functions, are available for *concrete evaluation* but are unavailable for standard *abstract evaluation* by virtue of their code being unavailable. Therefore we *concretely evaluate* the expressions and propagate the concrete objects through the points-to graph. This yields more information during inclusion constraint resolution and improves program coverage.

We evaluate PoTo on concrete type inference and call graph construction. We choose type inference because there has been significant interest in the problem in recent years. In addition to traditional approaches such as Pytype [12], there has been significant advance in deep-learning-based approaches, e.g., [1, 30, 46]. PoTo+ is the type inference client that largely draws types from PoTo's result: given $Pt(x)$ computed by the points-to analysis, the type of x is the union of the types of objects $o \in Pt(x)$. We compare PoTo+ against (1) Pytype [12], arguably the most advanced rule-based type inference tool, and (2) DLInfer [46], a state-of-the-art deep-learning-based tool, on a benchmark suite of Python packages ranging from 3,556 to 285,515 lines of code. Our results show that PoTo+ is comparable to Pytype and both techniques outperform DLInfer in terms of coverage (i.e., percentage of variables for which a type is reported) and correctness of inferred types. Furthermore, PoTo scales better than Pytype. Call graph construction is a classical client of points-to analysis. We build PoToCG based on PoTo and compare it to PyCG [34], the leading call graph construction analysis for Python. The results show that PoToCG constructs more complete and more precise call graphs compared to PyCG and it scales better.

The contributions of our work are as follows:

- PoTo, the first Andersen-style points-to analysis for Python.
- Hybridization weaving concrete and abstract evaluation.
- Evaluation on two clients of points-to analysis: type inference and call graph construction.

Ultimately, we hope that this work contributes to tools that make Python developers more productive and Python code more robust.

2 Problem Statement and Overview

The problem at hand is to design an Andersen-style points-to analysis for Python and *scale the analysis to large real-world Python packages.* Andersen's points-to analysis is a classical static analysis problem. It is a whole-program flow-insensitive and context-insensitive analysis that tracks flow of values via inclusion constraints. E.g., an assignment statement `x = y` triggers an inclusion constraint $Pt(\mathtt{y}) \subseteq Pt(\mathtt{x})$ indicating that the points-to set of `y` flows to the points-to set of `x`.

There are numerous challenges for Andersen's analysis and more generally static analysis for Python. First, Andersen's analysis is defined over a 3-address code intermediate representation (IR) serving as the foundation for the inclusion constraints. Translation from high-level C, C++, Java, and JavaScript code into 3-address code is well-studied and there are mature tools that provide the translation, notably LLVM [20], Soot [41], WALA [17], and Doop [36]. Surprisingly, static analysis for Python is largely ad-hoc AST-based analysis with each new work (e.g., [12, 46]) embedding its own interpretation of Python AST constructs and translation (if any) of those constructs into a 3-address code IR. Translating Python AST constructs into a 3-address code IR is challenging due to (1) Python's complex syntax and dynamic semantics and (2) Python's rich module system and scoping issues arising from it. Python allows for reflection and metaprogramming, subclassing conditioned on runtime values, complex With statements and context managers, and many other features [47] that are difficult or impossible to handle soundly in a static semantics. Even a construct such as Subscript, e.g., `a[index_expr]`, is non-trivial in the sense that its canonical interpretation as element access of a list-like structure is unsound (we choose the canonical interpretation for both Subscript and Attribute; a more faithful but also more complex interpretation would be to treat `__getitem__` and `__getattr__` as virtual calls). A principled analysis therefore must choose a subset of constructs to interpret precisely. We present a minimal syntax that specifies a subset of Python, and for that subset we present a precise interpretation that is faithful to the Python construct's semantics. All other Python AST constructs are abstracted by a fall-through structure *Other* and receive a default interpretation that can be both over-approximate and unsound. Going forward we sometimes refer to precisely-interpreted constructs as *interpreted* and to *Other* as *uninterpreted*. A key goal is to describe the points-to analysis as completely as possible, grounding the syntax and specifying interpretation semantics for both kinds of constructs, while at the same time abstracting away unnecessary detail. Section 3 specifies interpreted and uninterpreted constructs and their semantics, and makes it clear that we support memory-accesses and nested functions. It also explicitly lists the features under *Other*, e.g., generator expressions and With statements.

Another challenge is use of external libraries. Whole-program static analysis generally assumes that library code is available. However, Python has a particularly extensive ecosystem of open-source libraries; it is a dynamically-typed language where library interfaces usually lack type annotations; Python libraries and their interfaces are often rapidly evolving; and libraries are often at least partially implemented in C instead of Python. We address this with a novel hybridization technique that weaves concrete and abstract evaluation during 3-address code generation and constraint resolution.

While to the best of our knowledge we are the first to present a points-to analysis for Python, the problem has been studied extensively in mainstream languages, notably C, C++, Java, and JavaScript. (We detail related work in Section 6.) JavaScript is a dynamically typed language, as is Python. However, JavaScript poses unique challenges and program analysis aims to address these challenges: event-driven nature of applications, interaction with the DOM, runtime construction of object property strings, and runtime creation and deletion of object properties. None of these is a significant issue for Python, which increasingly is a general-purpose language used to build libraries and applications similarly to Java. We therefore model Andersen's analysis after Java's analysis, extracting and analyzing a Java-like subset of Python. Python presents new challenges though, including lack of a 3-address code IR, missing external libraries, and first-class functions and classes.

Consider the example in Figure 1, adapted from DLInfer, a recent paper on neural type inference for Python. Our analysis has two core phases (PoTo), followed by a client phase:

Phase 1: Python source ⟶ 3-address code
Phase 2: 3-address code ⟶ Points-to graph

```
1  import re
2
3  def url_regex():
4    regex_cache = None
5    if regex_cache is None:
6      regex_cache = re.compile(r"p")
7    return regex_cache
8
9  def str_validator(value):
10   if isinstance(value,str):
11     return value.value
12   else:
13     return value
14
15 def validate(value):
16   url = str_validator(value)
17   m = url_regex().match(url)
18   if m.end() != len(url):
19     raise Exception(m.end())
20
21 def main():
22   validate("p abcd")
```

(a) Python source

```
1  import re # External import
2
3  def url_regex():
4    regex_cache = None
5    regex_cache=(const,<class 're.Pattern'>,...)
6    url_regex_ret = regex_cache
7
8  def str_validator(value):
9    str_validator_ret = value.value
10   str_validator_ret = value
11
12 def validate(value):
13   url = str_validator(value)
14   t1 = url_regex()
15   t2 = t1.match
16   m = t2(url)
17   t3 = m.end
18   t4 = t3()
19   t5 = len(url)
20   t6 = (const,<class 'Exception'>,...)
21   t7 = m.end
22   t8 = t7()
23   t9 = t8(t7)
```

(b) 3-address code for (a), excluding `main`

Figure 1 Illustrating example, adapted from DLInfer [46].

1.	x = object	*new assignment*
2.	x = y	*copy propagation*
3.	x.f = y	*field write*
4.	x = y.f	*field read*
5.	x = y(z)	*closure call*

Figure 2 3-address code statements.

Phase 3 (client): Points-to graph ⟶ Concrete type inference or call graph

Phase 1 takes as input Python source code and produces 3-address code; this phase works at the granularity of a function. Phase 2 processes the 3-address statements as inclusion constraints computing the points-to graph. The two phases are intertwined — roughly, the analysis starts at the main function and immediately invokes Phase 1 on main generating 3-address code for main; it then invokes Phase 2 to solve the 3-address code for main. As new functions become reachable, the analysis invokes 3-address code generation, then it proceeds to solve the constraints until the points-to graph reaches a fixpoint. As expected, one can implement many client analyses on top of the points-to results. Phase 3 focuses on concrete (i.e., non-polymorphic) type inference and call graphs as client analyses.

The first phase does principled translation of Python AST constructs into the standard 3-address statements shown in Figure 2. For example, the Python source in Figure 1(a) Line 17 translates into the 3-address sequence of statements in Figure 1(b) Lines 14–16. A novelty in our treatment is the weaving of concrete evaluation into 3-address code translation and later constraint resolution. During translation, the analysis concretely evaluates every expression in its enclosing import environment. If evaluation succeeds, translation returns the resulting constant; otherwise, it proceeds recursively and returns the corresponding 3-address code statements. For example, in Figure 1(a) Line 6, the right-hand-side evaluates into a concrete object: `(const,<class 're.Pattern'>,...)` in Figure 1(b) Line 5 (we write a concrete object as a triple of `const`, the object type, and its value, with the value typically elided). As is standard for flow-insensitive analysis, the translation ignores control flow and basic blocks. Each function in Figure 1(b) is a straight-line sequence of 3-address statements. Suffix `_ret` indicates the function's return value (e.g., `str_validator_ret`).

The second phase of the analysis solves the 3-address statements as inclusion constraints, incorporating (1) class and function objects as first-class values, and (2) concrete evaluation. The analysis maintains *abstract objects* (constructed from code in the package under analysis) and *concrete objects* (constructed from concrete evaluation). Abstract objects, in turn, fall into three categories: *meta-class* objects, *meta-func* objects (more precisely closure objects), and *data* objects. The analysis maintains an abstract reference environment, which is the package-under-analysis environment, and uses it to resolve names defined in the package.

When processing a call or a field access statement, the analysis retrieves each object in the points-to set of the receiver variable. Roughly speaking, if the function object at the call is an abstract one, the analysis proceeds with abstract evaluation. If it is a concrete one, it attempts concrete evaluation. For example, in `url = str_validator(value)`, the analysis examines the points-to set of `str_validator` and retrieves the abstract function object representing the `str_validator` function. This triggers abstract evaluation and (standard) addition of the points-to set of `str_validator_ret` (the special return variable) to the points-to set of the left-hand-side variable `url`. Eventually, the points-to set of `url` becomes $\{$`(const, <class 'str'>, 'p abcd')`$\}$. Concrete evaluation of `t2 = t1.match` returns the concrete closure object corresponding to the match function, and finally concrete evaluation of `m = t2(url)` returns a concrete `re.Match` object. The hybrid analysis infers concrete types for identifiers `url` and `m`, respectively `<class 'str'>` and `<class 're.Match'>`, while Pytype reports only the fall-through type Any for both.

3 Syntax and Semantics

Recent work defines syntax for a subset of Python and a corresponding interpretation semantics for the purpose of weakest precondition inference [32]. A notable idea in this work is the separation of Python constructs into interpreted and uninterpreted ones. In this paper, we follow the idea of separating constructs into interpreted and *Other* but differ in important ways as our goal is interprocedural flow- and context-insensitive points-to analysis rather than weakest precondition inference. Unlike [32], we track object creation and flow of values and construct a call graph on the fly. Flow-insensitive points-to analysis demands a different set of interpreted constructs and a different interpretation.

Section 3.1 defines the syntax for the purposes of flow-insensitive points-to analysis, Section 3.2 defines the interpretation that generates 3-address code, and Section 3.3 presents Andersen-style constraint resolution, highlighting Python-specific semantics.

3.1 A Minimal Python Syntax

Figure 3 specifies the syntax which grounds the analysis. An expression e can be a constant c (`42` or `"foo"`), a variable (`x`), an attribute access (`x.foo`), a subscript access (`x["bar"]`), a list, a tuple, a dictionary, a list comprehension (`[2*x for x in range(10)]`), a dictionary

<table>
<tr><td>$e ::= c \mid x \mid e.x \mid e[e] \mid e(e, ..., e)$</td><td>const | Name | Attribute | Subscript | Call</td></tr>
<tr><td>$\quad \mid [e, ..., e] \mid \{e\text{: } e, ..., e\text{: } e\} \mid (e, ..., e)$</td><td>List | Dictionary | Tuple</td></tr>
<tr><td>$\quad \mid [e \text{ for } x, ..., x \text{ in } e \text{ if } e]$</td><td>ListComp</td></tr>
<tr><td>$\quad \mid \{e\text{:}e \text{ for } x, ..., x \text{ in } e \text{ if } e\}$</td><td>DictComp</td></tr>
<tr><td>$\quad \mid e \ op \ e \mid e \ cop \ e$</td><td>BinOp | Compare</td></tr>
<tr><td>$\quad \mid \mathit{Other} \ (e, ..., e)$</td><td><i>All other Python AST expressions</i></td></tr>
<tr><td>$s ::= \text{pass} \mid x = e \mid e.x = e \mid e[e] = e$</td><td>Pass | Assign</td></tr>
<tr><td>$\quad \mid s \ ; \ s \mid \text{for } e \text{ in } e\text{: } s$</td><td>Suite | For</td></tr>
<tr><td>$\quad \mid \text{def } f(x, ..., x)\text{: } s \ ; \ \text{return } e$</td><td>FunctionDef | Return</td></tr>
<tr><td>$\quad \mid \text{class } C(e, ..., e)\text{: } s$</td><td>ClassDef</td></tr>
<tr><td>$\quad \mid \mathit{Other} \ (s, ..., s)$</td><td><i>All other Python AST statements</i></td></tr>
<tr><td>$i ::= \text{import } p \ (\text{as } x)? \mid \text{from } p \text{ import } x \ (\text{as } x)?$</td><td>Import | ImportFrom</td></tr>
<tr><td>$\quad \mid i \ ; \ i$</td><td></td></tr>
<tr><td>$m ::= i \ ; \ s$</td><td>Module</td></tr>
</table>

Figure 3 Syntax of a subset of Python. The non-terminal names on the right are based on the official Python abstract syntax tree (AST); e.g., the AST node for a statement sequence is Suite.

comprehension (`{x:f(x)for x in range(x)}`), a binary operation or a compare operation. All other Python expressions are under *Other* $(e_1, ..., e_n)$ and receive the default interpretation; their AST nodes are BoolOp, NamedExpr, UnaryOp, Lambda, IfExp, Set, SetComp, GeneratorExp, Await, Yield, YieldFrom, FormattedValue, JoinedStr, Starred, and Slice.

A statement s can be pass, the assignment of either a variable (`x = 42`), an attribute (`x.foo = 42`), or a subscript (`x["bar"] = 42`), a sequence, a loop, a function definition, or a class definition. All other Python statements are captured under *Other* $(s_1, ..., s_n)$ and receive the default interpretation; their AST nodes are Delete, TypeAlias, AugAssign, AnnAssign, AsyncFor, While, If, With, AsyncWith, Match, Raise, Try, TryStar, Assert, Global, Nonlocal, Break, and Continue. This default interpretation is often all that is needed. For example, Break and Continue are irrelevant to flow-insensitive analysis. As another example, for If statements, the default interpretation descends into the AST node of the children and processes them. As illustrated in Figure 1, this is adequate for flow-insensitive analysis. To simplify the presentation, we leave out the syntax of complex left-hand-side expressions in assignments (e.g., `a, *(b, c)= (1, (2, 3), 4)`), variable-length arguments, and keywords arguments. We use deconstruct-and-bind preprocessing to reduce these into sequences of assignments as in Figure 3. For example, `a, *(b, c)= (1, (2, 3), 4)` reduces into a sequence including `a = 1`, `b = (2, 3)` and `c = 4`.

A module m starts with a sequence of import i, import p or from p import x, with an optional alias name (as x), followed by a statement. A package under analysis is a sequence of modules.

3.2 3-address Code Generation

3-address-code translation takes as input the Python source (i.e., AST) of a function and produces a sequence of 3-address-code statements of the form in Figure 2.

```
# Initialize $\Gamma_0$:
$\Gamma_0 = [], \Phi = \{\}$
for $\langle$module $M$ : $i; s\rangle$ in package under analysis
    for $\langle$class $C(...)$: ...$\rangle$ in $s$: $\Gamma_0 \leftarrow [(M.C, \mathsf{t})] + \Gamma_0$, t is fresh
    for $\langle$def $f(...)$: ...$\rangle$ in $s$: $\Gamma_0 \leftarrow [(M.f, \mathsf{t})] + \Gamma_0$, t is fresh
    # Imports from $p$ import $x'$ as $x$ are implicit assignments
    for $x = ...$ in $s$: $\Gamma_0 \leftarrow [(M.x, \mathsf{t})] + \Gamma_0$, t is fresh
# Next, compute class hierarchy $H$ and MRO:
$H \leftarrow C3(\Gamma_0)$ # $H$ maps ($\langle$class $C(...)$: ...$\rangle$, $f$) to $\langle$def $f(...)$: ...$\rangle$
# Interpret main and add to worklist:
$\Phi[\langle$def $main(...)$: $s\rangle] \leftarrow \mathcal{I}(s, [])$
$W \leftarrow \{\langle$def $main(...)$: $s\rangle\}$ # Entry point
# Interpret module initializers and add to worklist:
for $\langle$module M : ...$\rangle$ in package under analysis
    $\Phi[\langle$def $M.module\_init(...)$: $i; s\rangle] \leftarrow \mathcal{I}(i; s, [])$
    $W \leftarrow \{\langle$def $M.module\_init(...)$: $i; s\rangle\}$ # Entry points
# Solve constraints in reachable functions:
while $W \neq \emptyset$
   $\langle$def $f(...)$: ...$\rangle \leftarrow$ remove function from $W$
    for $c$ in $\Phi[\langle$def $f(...)$: ...$\rangle][1]$: $W \leftarrow W + c.solve()$
```

Figure 4 Worklist algorithm.

3.2.1 Environment and Interpretation Functions

The analysis interprets expressions and statements in an *abstract* reference environment split into two components: Γ and Γ_0. Γ is a *local* reference environment associated to the enclosing function. It is a map of (id,t) tuples where id is a local and t is the *analysis variable* representing the local. Γ_0 is the *global* reference environment mapping identifiers for module-level constructs to analysis variables representing those constructs. In addition, there is an *external* environment, Γ_{ext}, necessary for concrete evaluation. We analyze imports and group them into two categories: (1) internal imports are relative imports (e.g., `from ..metrics import get_scorer`) and ones referencing the analysis package, and (2) the remaining external imports (e.g., `import re`). External imports comprise Γ_{ext}.

The interpretation of a statement $\mathcal{I}(s, \Gamma)$ returns a pair (Γ', S) where Γ' is the augmented local environment resulting from the interpretation of s into 3-address code, and S is *the sequence of 3-address statements* corresponding to s. E.g., $\mathcal{I}(\mathsf{x} = \mathsf{y.f.g}, [(\mathsf{y}, \mathsf{t1})])$ returns the augmented environment [(x,t2), (y,t1)] and the following sequence of 3-address statements: t4 = t1.f; t3 = t4.g; t2 = t3. Here t1 and t2 are the analysis variables associated with local variables y and x respectively (in code examples we sometimes ignore analysis variables and use the source-level identifier directly). We leave Γ_0 and Γ_{ext} implicit in the writeup as they are uniquely defined for each statement under interpretation.

The interpretation of an expression $\mathcal{I}(e, \Gamma)$ returns a pair (V, S) where V is *a set of analysis variables* and S is the sequence of 3-address statements corresponding to e. E.g., $\mathcal{I}(\mathsf{y.f.g}, [(\mathsf{y}, \mathsf{t1})])$ returns fresh variable t3 and the sequence of 3-address statements t4 = t1.f; t3 = t4.g.

Φ is the environment of interpreted functions: a mapping from a function definition (Python AST FunctionDefs in the implementation) to a pair (Γ, S), where Γ is the local reference environment resulting from the translation of the function definition and S is the sequence of 3-address statements corresponding to the function body.

The analysis uses the worklist algorithm in Figure 4.

3.2.2 Global Environment Initialization

The analysis first initializes a global environment Γ_0 that contains mappings for identifiers of module-level constructs, $M.f$ (functions), $M.C$ (classes), and $M.x$ (identifier definitions) to their corresponding analysis variables. The scope of these constructs spans the entire package, hence the analysis augments the environment and makes these constructs available during 3-address code generation. This is similar to the *let rec* construct in functional programming which extends the scope for let-bound identifiers across all right-hand-side expressions. Initialization does not interpret functions, classes and right-hand-side of assignments; these are interpreted during reachability analysis.

3.2.3 Class Hierarchy Analysis

Next, the analysis computes the class hierarchy H using the C3 linearization algorithm for method resolution with multiple inheritance [4]. H is a mapping from a pair of class definition $\langle$class $C(...)$: ...$\rangle$ and function name f to the corresponding function definition $\langle$def $f(...)$: ...$\rangle$ resulting from a lookup in the *Method Resolution Order* (MRO) of C. Additionally, the step creates a (meta-cls, $\langle$class $C(...)$: ...$\rangle$) object for each module-level class definition and associates the corresponding analysis variable t to that object. In other words, this step creates an initial set of points-to edges which allows for data object creation during the second phase of the analysis.

3.2.4 Iteration

As it is customary for whole-program analysis, the analysis starts from a main function. It translates main and all module initializers into 3-address code (the calls to $\mathcal{I}(s, [])$) and adds them to the worklist. The analysis removes a function from the worklist and processes the 3-address statements c for that function. The call $c.solve()$ solves the semantic constraints associated to c which has side effects on the 3-address code environment Φ and points-to graph Pt. A call statement triggers interpretation and placement on the worklist of the callee function and it is an invariant that when a function is removed from the worklist, its three address code is available in Φ.

The call $c.solve()$ returns a minimal set of functions that are affected by solving constraint c. For example, if c is the 3-address code statement t1 $=$ t2 and solving it changes the points-to graph of t1, *solve* returns the enclosing function of the statement. A change in the points-to graph due to t1.f $=$ t2 returns all functions in Φ, as the effect of new objects added to Pt(o.f), where o is an object in the points-to set of t1, may propagate to arbitrary functions.

3.2.5 Interpretation of statements and expressions

The definition of the interpretation function is presented in Figures 5 and 6. Interpretation of statements is in the abstract while interpretation of expressions weaves concrete and abstract evaluation. Below we highlight the most interesting points.

3.2.5.1 Statements

Consider interpretation of assignment $x = e$. If the variable x appears in a module initializer, we retrieve the analysis variable from Γ_0 (imports also create implicit global assignments). Otherwise, we only augment the local environment with a fresh variable if the variable x is not in the current environment. We target flow-insensitive points-to analysis, and thus, a

```
$\mathcal{I}(\text{pass}, \Gamma)$                 =  return $(\Gamma, \{\})$

$\mathcal{I}(x = e, \Gamma)$                =  $(R, S) \leftarrow \mathcal{I}(e, \Gamma)$
                                   if scope is $M.module\_init$ :
                                     $\mathsf{t} \leftarrow lookup(M.x, \Gamma_0)$
                                     return $(\Gamma, S \cup \{\mathsf{t} = t_e \mid t_e \in R\})$
                                   if $x \in \Gamma$ :
                                     $\mathsf{t} \leftarrow lookup(x, \Gamma)$
                                     return $(\Gamma, S \cup \{\mathsf{t} = t_e \mid t_e \in R\})$
                                   else:
                                     $\mathsf{t} \leftarrow$ fresh variable
                                     $\Gamma' \leftarrow [(x, \mathsf{t})] + \Gamma$
                                     return $(\Gamma', S \cup \{\mathsf{t} = t_e \mid t_e \in R\})$

$\mathcal{I}(s_1 \; ; \; s_2, \Gamma)$              =  $(\Gamma_1, S_1) \leftarrow \mathcal{I}(s_1, \Gamma)$
                                   $(\Gamma_2, S_2) \leftarrow \mathcal{I}(s_2, \Gamma_1)$
                                   return $(\Gamma_2, S_1 \cup S_2)$

$\mathcal{I}(\text{for } x \text{ in } e\text{: } s, \Gamma)$        =  $\mathcal{I}(x = e \; ; \; s, \Gamma)$

$\mathcal{I}(\text{def } f(args)\text{: } body, \Gamma)$    =  if scope is $M.module\_init$ :
                                     $\mathsf{t} \leftarrow lookup(M.f, \Gamma_0)$
                                     return $(\Gamma, \{\mathsf{t} = (\text{meta-func}, \langle \text{def } f(args)\text{: } body \rangle)\})$
                                   else:
                                     $\mathsf{t} \leftarrow$ fresh variable
                                     $\Gamma' \leftarrow [(f, \mathsf{t})] + \Gamma$
                                     return $(\Gamma', \{\mathsf{t} = (\text{meta-func}, \langle \text{def } f(args)\text{: } body \rangle)\})$

$\mathcal{I}(\text{class } C(args)\text{: } body, \Gamma)$  =  return $(\Gamma, \{\})$

$\mathcal{I}(Other \; (s_1, ..., s_n), \Gamma)$     =  $(\Gamma_1, S_1) \leftarrow \mathcal{I}(s_1, \Gamma)$
                                   ...
                                   $(\Gamma_n, S_n) \leftarrow \mathcal{I}(s_n, \Gamma_{n-1})$
                                   return $(\Gamma_n, S_1 \cup ... \cup S_n)$
```

Figure 5 From Python statements to 3-address-code. Given an environment Γ, the interpretation function for a statement $\mathcal{I}(s, \Gamma) = (\Gamma', S)$ returns an updated environment Γ' and the 3-address code S. M is the enclosing module, and Γ_0 is the global environment.

```
$\mathcal{I}(x, \Gamma)$  =  if $x \in \Gamma$: return $(\{lookup(x, \Gamma)\}, \{\})$
                              if $M.x \in \Gamma_0$: return $(\{lookup(M.x, \Gamma_0)\}, \{\})$
                              if $(\mathsf{const}, ty, v) \leftarrow eval(x, \Gamma_{ext})$:
                                 return $(\{x\}, \{x = (\mathsf{const}, ty, v)\})$
                              else: return $(\{\}, \{\})$

$\mathcal{I}(e.f, \Gamma)$  =  $\mathsf{t} \leftarrow$ fresh variable
                              if $(\mathsf{const}, ty, v) \leftarrow eval(e.f, \Gamma_{ext})$:
                                 return $(\{\mathsf{t}\}, \{\mathsf{t} = (\mathsf{const}, ty, v)\})$
                              $(V, S) \leftarrow \mathcal{I}(e, \Gamma)$
                              return $(\{\mathsf{t}\}, S \cup \{\mathsf{t} = t_e.f \mid t_e \in V\})$

$\mathcal{I}(e_1[e_2], \Gamma)$  =  $\mathsf{t} \leftarrow$ fresh variable
                              if $(\mathsf{const}, ty, v) \leftarrow eval(e_1[e_2], \Gamma_{ext})$:
                                 return $(\{\mathsf{t}\}, \{\mathsf{t} = (\mathsf{const}, ty, v)\})$
                              $(V_1, S_1) \leftarrow \mathcal{I}(e_1, \Gamma)$
                              $(V_2, S_2) \leftarrow \mathcal{I}(e_2, \Gamma)$
                              return $(\{\mathsf{t}\}, S_1 \cup S_2 \cup \{\mathsf{t} = t_1[] \mid t_1 \in V_1\})$

$\mathcal{I}(e_1(e_2), \Gamma)$  =  $\mathsf{t} \leftarrow$ fresh variable
                              if $(\mathsf{const}, ty, v) \leftarrow eval(e_1(e_2), \Gamma_{ext})$:
                                 return $(\{\mathsf{t}\}, \{\mathsf{t} = (\mathsf{const}, ty, v)\})$
                              $(V_1, S_1) \leftarrow \mathcal{I}(e_1, \Gamma)$
                              $(V_2, S_2) \leftarrow \mathcal{I}(e_2, \Gamma)$
                              return $(\{\mathsf{t}\}, S_1 \cup S_2 \cup \{\mathsf{t} = t_1(t_2) \mid t_1 \in V_1, t_2 \in V_2\})$

$\mathcal{I}([e], \Gamma)$  =  $\mathsf{t} \leftarrow$ fresh variable
                              if $(\mathsf{const}, ty, v) \leftarrow eval([e], \Gamma_{ext})$:
                                 return $(\{\mathsf{t}\}, \{\mathsf{t} = (\mathsf{const}, ty, v)\})$
                              $(V, S) \leftarrow \mathcal{I}(e, \Gamma)$
                              $S' \leftarrow \{\mathsf{t} = (\mathsf{data}, \langle \mathsf{class\ list} \rangle)\} \cup \{\mathsf{t}[] = t_e \mid t_e \in V\}$
                              return $(\{\mathsf{t}\}, S \cup S')$

$\mathcal{I}(Other\ (e_1, ..., e_n), \Gamma)$  =  if $(\mathsf{const}, ty, v) \leftarrow eval(Other\ (e_1, ..., e_n), \Gamma_{ext})$:
                                 $\mathsf{t} \leftarrow$ fresh variable
                                 return $(\{\mathsf{t}\}, \{\mathsf{t} = (\mathsf{const}, ty, v)\})$
                              else:
                                 $(V_1, S_1) \leftarrow \mathcal{I}(e_1, \Gamma)$
                                 ...
                                 $(V_n, S_n) \leftarrow \mathcal{I}(e_n, \Gamma)$
                                 return $(V_1 \cup ... \cup V_n, S_1 \cup ... \cup S_n)$

$\mathcal{I}(\mathsf{from}\ p\ \mathsf{import}\ x'\ \mathsf{as}\ x, [])$  =  $M' \leftarrow$ find module of $x'$
                              $\mathsf{t1} \leftarrow lookup(M'.x', \Gamma_0)$
                              $\mathsf{t2} \leftarrow lookup(M.x, \Gamma_0)$
                              return $(\{\}, \{\mathsf{t2} = \mathsf{t1}\})$
```

Figure 6 From Python expressions and imports to 3-address-code. Given an environment Γ, the interpretation function for an expression $\mathcal{I}(e, \Gamma) = (V, S)$ returns a set of analysis variables V and the 3-address code S. M is the expression's enclosing module, and Γ_{ext} is its external environment. Γ_0 is the global environment.

Python sequence `x = 1; x = "a"` gives rise to 3-address code sequence t1 = 1; t1 = "a" and fails to distinguish that `x` is an integer at the first assignment and a string at the second.

The interpretation of a loop for x in e: s reduces to a sequence of assignment $x = e$ followed by s. The assignment binds identifier x before descending into the interpretation of s.

A function definition in a module initializer has an entry $M.f$ in Γ_0. Otherwise, i.e., if this function is nested into another function, we augment the local environment with f and return the augmented environment. Function definition gives rise to a "new" statement, assigning the abstract meta-func object to t. Class definitions have no effect during interpretation. Module-level class definitions are processed during class hierarchy analysis.

Note also that we ignore static reference environments for nested functions, which is unsound in general. A function value may flow to arbitrary points of the program and it is interpreted into 3-address code when it is called (detailed in calls in Section 3.3); however, interpretation happens in the empty environment rather than the actual static reference environment and references coming from enclosing scopes evaluate to empty sets.

For uninterpreted statements, the algorithm descends into each sub-statement and extracts the corresponding 3-address code. This does not "glue" components according to the statement's semantics. However, recursive descent processes all nested assignments and calls; it augments the environment and generates 3-address code that captures value flow.

3.2.5.2 Expressions

Recall that a key feature of our analysis is concrete evaluation and that concrete evaluation happens both during 3-address code generation and constraint resolution. We employ the following heuristic for 3-address code generation, as the rules for expression interpretation in Figure 6 detail. For simple expressions (i.e., variables), we first attempt resolution in the abstract environment: first Γ then Γ_0. If it fails, we attempt concrete evaluation. For all other expressions (i.e., complex expressions), we first attempt concrete evaluation in Γ_{ext}, and if it fails to produce a concrete object we proceed with interpretation in the abstract.

The interpretation of a variable (first rule in Figure 6) first searches the local environment Γ (i.e., the enclosing function), and then the global environment Γ_0 if the first lookup fails. If lookup in Γ_0 fails as well, it attempts concrete evaluation. If concrete evaluation succeeds and returns a concrete object, $\mathcal{I}(x, \Gamma)$ returns x and an object creation statement. The last clause aims at built-in functions, e.g., `len` and enables reporting of built-in callees. The reader may have noticed here that we reuse the original variable name as an analysis variable, but create a new concrete instance of the built-in function object. Thus, there are as many concrete instances representing `len` as there are (static) calls to `len` in the code. Our implementation does not optimize this case yet, but it is trivial to get rid of such redundant concrete objects.

For remaining expressions, including *Other*, the analysis first tries concrete evaluation, and if it succeeds, it returns the new analysis variable t along with the assignment of the concrete object to t. If an expression does not evaluate (which is the common case), interpretation continues recursively in the abstract environment. For example, `np.array(arg)` matches Call in Figure 6. Concrete evaluation fails with a NameError as `arg` is a local program variable; interpretation proceeds in the abstract. It recurses into Attribute `np.array` and this time it does evaluate to the numpy function. Interpretation of the call expression returns ({t1}, {t2 = (const, <... builtin ...>,...); t1 = t2(t3)}) where t3 is the analysis variable for `arg`.

Subscript expressions treat [] as a special field in the abstract, which is standard in points-to analysis. To interpret a list in the abstract, we create a new list object and generate subscript assignments to populate the list. Tuple, set, and dictionary are analogous.

For uninterpreted expressions we return the union resulting from the interpretation of each sub-expression when concrete evaluation fails.

3.2.5.3 Import

Finally, to interpret imports in module initializers (last rule of Figure 6), we first find the enclosing module M' of imported construct x', and then look up for the representative analysis variables t1 and t2 corresponding to the imported construct $M'.x'$ and the alias x in the current module M. Statement t2 = t1 propagates the value (e.g., a function definition, a class definition) from module M' to M. For example, consider statement `from cerberus.platform import _get_args as get_args` in module M. Lookup of `cerberus.platform._get_args` yields t1 which points to the function def of _get_args. Lookup of M.get_args yields t2 and t2 = t1 ensures the def propagates to t2 and references to get_args in M find that function def.

For simplicity, the rule treats x' as a module-level construct, however, it can be a module. E.g., `errors` in `from cerberus import errors` is a module and code can access module-level constructs as in `errors.DocumentErrorTree`. We handle this case in the attribute rule: if lookup of e in Γ during $\mathcal{I}(e, \Gamma)$ yields no result, the analysis maps $e.f$ to a module-level-construct identifier $M.C$, $M.f$ or $M.x$, and searches Γ_0 to retrieve the corresponding analysis variable (the step is not shown in the rule to avoid the complexity of lookup failure).

3.3 Constraint Resolution

As mentioned earlier, the analysis maintains abstract objects and concrete objects. Abstract objects are explicitly grouped into *data objects*, *function objects*, and *class object*:

(data, ⟨class C(...): ...⟩)	*an abstract data object*
(meta-func, ⟨def f(...): ...⟩)	*an abstract function object*
(meta-cls, ⟨class C(...): ...⟩)	*an abstract class object*
(const, ty, v)	*a concrete object*

The 3-address statements are as specified in Figure 2. These statements induce constraints that populate a points-to graph Pt. The nodes in the points-to graph are analysis variables as well as objects o of the above kinds. The edges represent the points-to relation. E.g., there could be an edge from variable t to an object o, and this is denoted as $\{o\} \subseteq Pt(t)$ or equivalently as $t \rightarrow o$. There could be a field edge indicating that field f of o_1 points to o_2 and this is denoted as $\{o_2\} \subseteq Pt(o_1.f)$ or equivalently as $o_1 \xrightarrow{f} o_2$.

The rules for new assignment, copy propagation and field write are largely standard (except for their returns) and we elide them from the presentation. We elaborate on the rules for field read $t_1 = t_2.f$ and function call $t_1 = t_2(t_3)$ as they illustrate Python-specific semantics and concrete evaluation.

3.3.1 Indirect read

The rule in Figure 7 examines each object in the points-to set of receiver variable t_2 and does case-by-case analysis on the object type. E.g., if it is an abstract data object of some user-defined class, the analysis searches the class hierarchy (MRO) to determine the function referenced by $o.f$, forms the closure by binding *self* to the receiver object o and adds the closure object to the points-to set of left-hand-side t_1. If the object is a concrete object, the analysis evaluates the field access returning a new concrete object and adding it to the points-to set of t_1. Here, and in Calls, we restrict the number of times a statement is

```
$solve$ for $t_1 = t_2.f$ in ⟨def $f'$(...): ...⟩ with $\Gamma_{ext}$ :
  for $o \in Pt(t_2)$
    case $o$ of
      (data, ⟨class $C$(...): ...⟩) $\rightarrow$
        ⟨def $f$(self, $p$): ...⟩ $\leftarrow H[($⟨class $C$(...): ...⟩$, f)]$
        $Pt(t_1) \leftarrow Pt(t_1) + \{$(meta-func, ⟨def $f(o, p)$: ...⟩)$\}$
      (meta-cls, ⟨class $C$(...): ...⟩) $\rightarrow$
        ⟨def $f$(self, $p$): ...⟩ $\leftarrow H[($⟨class $C$(...): ...⟩$, f)]$
        $Pt(t_1) \leftarrow Pt(t_1) + \{$(meta-func, ⟨def $f$(self, $p$): ...⟩)$\}$
      (const, ...) $\rightarrow Pt(t_1) \leftarrow Pt(t_1) + \{eval(o.f, \Gamma_{ext})\}$
    $Pt(t_1) \leftarrow Pt(t_1) + Pt(o.f)$ # when $f$ is an object field, add its points-to set
  return {⟨def $f'$(...): ...⟩} if $Pt(t_1)$ changed else {}
```

Figure 7 Indirect read. Solving $t_1 = t_2.f$ in enclosing function ⟨def f'(...): ...⟩ with Γ_{ext}.

evaluated successfully. The restriction, currently set to 2, is not only an optimization but also necessary for termination, otherwise the analysis would keep creating distinct concrete objects at the same statement during fixpoint iteration.

The last line returns the enclosing function f' to the worklist if there is a change to the points-to set of t_1, as the change may trigger changes to other points-to sets in f'.

3.3.2 Call

Figure 8 details the rule for calls. There are two cases at the top level, an abstract object or a concrete object as function value. For an abstract object, we do case-by-case analysis. If o is a data object, it queries the hierarchy to retrieve the corresponding __call__ function — this is the function that is being called. If it is a meta class object, this leads to the retrieval of the constructor. If it is a meta function, there are two cases: a closure where self is already bound to a receiver, or the function is just a value with a null reference environment.

Once the analysis identifies the function to be called at this site, it checks if an interpretation of this function into 3-address code already exists. If it does not we interpret it. Notably, we interpret the AST of the function in the empty environment (i.e., only parameters are bound). This means that a first-class function does not carry its static reference environment and the analysis introduces unsoundness. While it is possible to extend the analysis with such bindings, this will complicate the code, while we believe in practice it will have limited impact (based on looking at Python code and our results). A notable departure from standard Java analysis is that there is no explicit new A() site. Different meta class objects may flow to receivers of calls accounting for data object creation (but the number of meta class objects is bounded and thus the number of data objects instantiated at the call is finite).

Once the callee function is interpreted, the analysis propagates points-to sets from actual arguments t_3 to formal parameters p and return values to the left-hand-side of the call t_1. In case of a concrete receiver, if the analysis finds concrete arguments for all parameters, it executes the function.

4 Results

We evaluate our Andersen-style points-to analysis, PoTo, on two clients: type inference (Section 4.1) and call graph construction (Section 4.2). However, points-to analysis has a wide variety of applications and we envision other clients as well. The analysis can be run on any Python package. It starts at a provided entry function and computes a points-to graph

```
$solve$ for $t_1 = t_2(t_3)$ in $\langle$def $f'(...)$: ...$\rangle$ with $\Gamma_{ext}$ :
  for $o \in Pt(t_2)$
      if $o$ is an abstract object
        case $o$ of
          (data, $\langle$class $C(...)$: ...$\rangle$) $\rightarrow$ # $call on data/instance$
            $callee \leftarrow H[(\langle$class $C(...)$: ...$\rangle$, '__call__')]
            $rcv \leftarrow \{o\}$
          (meta-cls, $\langle$class $C(...)$: ...$\rangle$) $\rightarrow$ # $constructor call$
            $callee \leftarrow H[(\langle$class $C(...)$: ...$\rangle$, '__init__')]
            $rcv \leftarrow \{$(data, $\langle$class $C(...)$: ...$\rangle)\}$ # $new object$
          (meta-func, $\langle$def $f(o', p)$: $s\rangle$) $\rightarrow$ # $closure call$
            $callee \leftarrow \langle$def $f($self$, p)$: $s\rangle$
            $rcv \leftarrow \{o'\}$
          (meta-func, $\langle$def $f(p)$: $s\rangle$) $\rightarrow$
            $callee \leftarrow \langle$def $f(p)$: $s\rangle$ # $the function def$
            $rcv \leftarrow$ None
        $\langle$def $f(p)$: $s\rangle = callee$ # $match and deconstruct callee$
        if $\langle$def $f(p)$: $s\rangle \notin \Phi$ # $callee is not interpreted$
          t $\leftarrow$ fresh variable
          $\Phi[\langle$def $f(p)$: $s\rangle] \leftarrow \mathcal{I}(s, [(p, \mathsf{t})])$ # $env.$ $includes$ self $when$ $f$ $is an instance function$
        if $rcv \neq$ None # $there is a receiver$
          $t_4 \leftarrow \Phi[\langle$def $f($self$, p)$: $s\rangle][0][$self$]$ # $retrieve analysis variable corresponding to$ self
          $Pt(t_4) \leftarrow Pt(t_4) + rcv$ # $receiver to$ self
        $t_5 \leftarrow \Phi[\langle$def $f(p)$: $s\rangle][0][p]$
        $Pt(t_5) \leftarrow Pt(t_5) + Pt(t_3)$ # $actual to formal$
        $t_6 \leftarrow \Phi[\langle$def $f(p)$: $s\rangle][0][f$_ret$]$
        $Pt(t_1) \leftarrow Pt(t_1) + Pt(t_6)$ # $ret var to lhs of call$
      else # $o is a concrete object$
        for $o_1$ in $Pt(t_3)$
          if $o_1$ is a concrete object
            $Pt(t_1) \leftarrow Pt(t_1) + \{eval(o(o_1), \Gamma_{ext})\}$
  return $\{\langle$def $f(...)$: ...$\rangle, \langle$def $f'(...)$: ...$\rangle\}$ if change else $\{\}$
```

Figure 8 Call: solving for $t_1 = t_2(t_3)$ in enclosing function $\langle$def $f'(...)$: ...$\rangle$ with Γ_{ext}.

containing information on reachable variables and their inferred types. To thoroughly analyze a library package, a set of entry functions are needed. We follow DLInfer, a neural type inference for Python, and use the same 10 Python packages from their experiment available with DLInfer's artifact. The 10 packages range in size from 3,556 LOC to 285,515 LOC (see Table 1). These packages contain rich test cases which suit our need of diverse entry points. We use each function in a package's test directory as an entry point. For five of the packages (cerberus, mtgjson, pygal, sc2, and zfsp), we additionally create custom entry functions targeting remaining unreachable public functions. For the other five packages (anaconda, ansible, bokeh, invoke, and wemake_python_styleguide), we only use default test suites.

4.1 Type Inference

To target unreachable methods for the purposes of type inference, we enhance the analysis. We achieve this by performing a shallow analysis that collects built-in type information and type annotations from assignment and return statements. For example, consider an assignment with a built-in type: `rules = set(schema.get(field, ()))`. The shallow analysis infers that `rules` can be of type `set`. This has impact when the method is unreachable from

PoTo's entry points, as it creates a key and infers a type for the `rules` local variable. The final results, called PoTo+, are stored as a dictionary of *keys* to their inferred types. A *key* is a tuple of (module name, function name, variable name), describing the variable and its scoping information. Keys largely correspond to local variables (including arguments and returns) in a package and are an abstraction for flow-insensitive analysis, the target of our work. The remainder of this section uses the terms keys and variables interchangeably.

We compare the results of PoTo+ against four other type inference techniques. Three of these are based on the recent neural type inference work DLInfer [46]: DL-ST, DL-DY, DL-ML. We make use of the result files available with DLInfer's artifact [45]. DL-ST is their ground truth information, which is a combination of running the Pysonar2 static tool and extracting type information [42]. DL-DY is a set of dynamic type information obtained from executing the test suites. This dynamic set contains only variables whose type can be collected only this way, meaning that this is the set difference of the actual dynamic run and DL-ST set [46]. Lastly, DL-ML is the result of DLInfer's machine-learning approach. We aggregate each DLInfer result in the same way as our analysis, which is a dictionary of keys to their types. We choose DLInfer for several reasons: (1) DLInfer is recent work in a top conference, (2) it compares with several state-of-the-art deep-learning techniques: Type4Py [27], Typilus [1], PYInfer [9], and DeepTyper [15], (3) DLInfer infers types for local variables, not only parameters and returns, and (4) its artifact [45] includes full Python packages along with analysis results allowing for a comparison over the same code base.

In addition to DLInfer, we also compare our result to Pytype, a prominent static type checking and type inference tool [12]. To get type information for all variables, we instrument the package by inserting Pytype's command `reveal_type(var)` for each local variable at the end of a function, and run Pytype on each file in the package directory. The types are then collected and combined to be the inferred result of Pytype. This process takes time to complete but only needs to be done once. This is a departure from standard use of Pytype as baseline for type inference work, which uses Pytype's result on parameters and returns only (e.g. [29]); Pytype can infer types for local variables as well and we make use of this in our comparison.

Our evaluation considers 4 research questions:

RQ1: How high is the coverage of PoTo+ compared to other type inference techniques?
RQ2: To what extent are the types from PoTo+ equivalent to those from other techniques?
RQ3: If the types do not match those from other techniques, which one is correct?
RQ4: Does the time to run PoTo+ scale well and how does it compare to Pytype?

To answer RQ1, we measure the percentage of total keys in a package for which our analysis reports types against the percentage of total keys for which the four other type inference techniques report types (more detail on methodology in Section 4.1.1). Our analysis collects types for a *larger percentage of total keys* compared to the four other techniques. To answer RQ2, we compare our inferred types to those inferred by each of the four other techniques, measuring the equivalence in term of total match, partial match, and mismatch (more detail in Section 4.1.2). Our inferred types *largely match with Pytype's*. While there is some match with DLInfer, there is disagreement in many keys.

To answer RQ3, we inspect a sample of mismatches according to the result from RQ2 for each pair comparisons: PoTo+ vs. Pytype, PoTo+ vs. DL-*. We found that in the few cases where there is a mismatch with Pytype, Pytype is correct in all cases. A mismatch with DL-* is nearly always a correct result by PoTo but an incorrect result by DL-*. The conclusion from RQ1-RQ3 is that *traditional techniques outperform a state-of-the-art neural technique* for the task of type inference. Lastly, for RQ4, we compare the total time to run

Table 1 Statistics for the dataset.

Package	Files		LOC		Keys
	Total	Test Dir	Total	Test Dir	
cerberus	45	32	6,694	3,011	966
mtgjson	54	2	6,912	58	1,378
pygal	78	24	13,780	3,208	2,439
sc2	69	6	11,205	603	5,970
zsfp	54	7	3,556	207	1,345
anaconda	370	9	90,207	587	21,183
ansible	1,445	961	285,515	174,296	22,346
bokeh	1,133	280	131,931	43,316	14,978
invoke	133	65	26,159	9,878	3,474
wemake	403	291	55,841	32,798	3,184

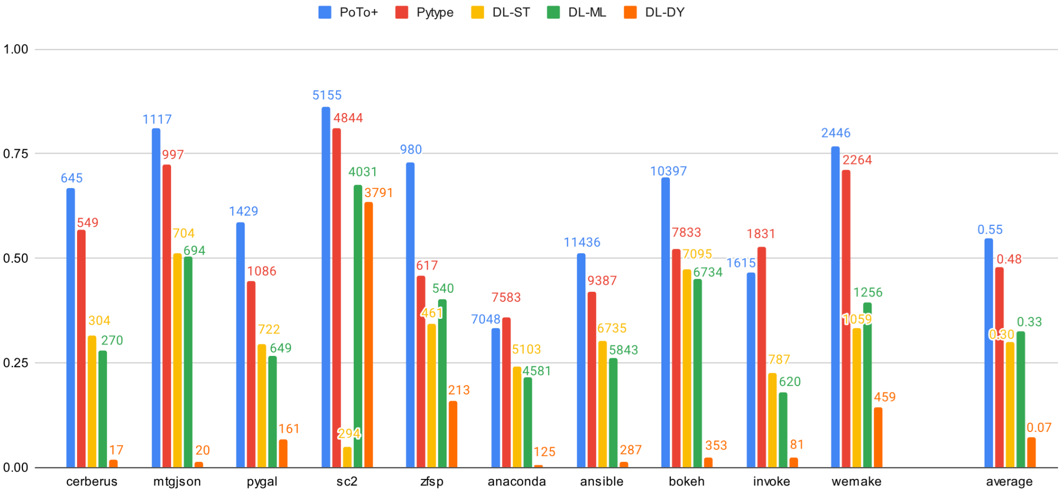

Figure 9 Coverage percentages of non-empty keys to total keys (RQ1).

PoTo (with dozens of entry functions for the smaller packages and thousands for the larger ones) to the time to run Pytype to collect reveal-type information. On all but one package, ansible, PoTo outperforms Pytype significantly.

4.1.1 Coverage (RQ1)

Table 1 shows the statistic of all 10 packages. We are interested in inferring types for the core package and exclude test files from our reports. The total keys include all variables, function arguments, and function return types. They serve as an upper limit of possible variables (or keys) for each package.

Figure 9 shows the percentages of non-empty keys to the total keys for each package. Non-empty keys are ones for which the analysis (ours or other techniques) gives type information; empty keys are variables in total keys for which the analysis gives no information. In the case of Pytype, a key is designated as empty if its inferred type is the trivial `Any` type. The numbers above each bar are the numbers of non-empty keys. For the five packages (cerberus, mtgjson, pygal, sc2, and zfsp) where we add custom entry functions, PoTo+ covers

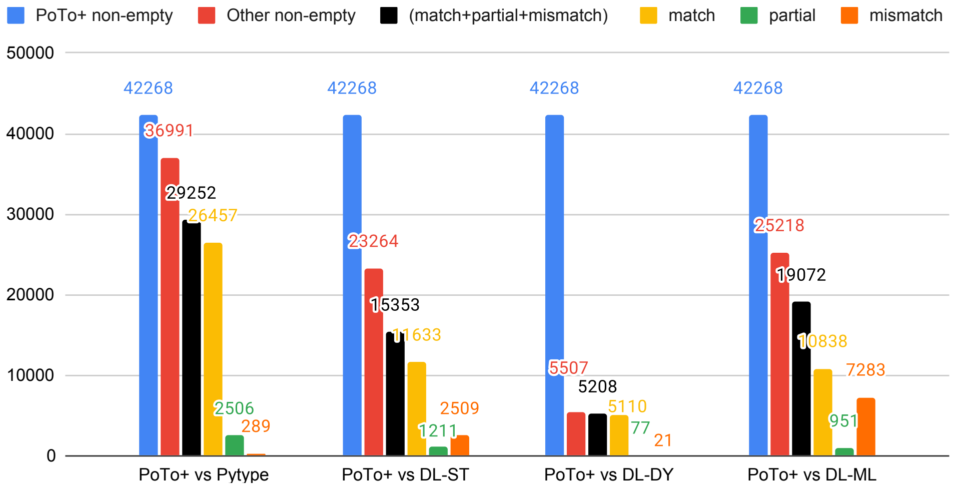

Figure 10 Equivalence comparison between PoTo+ and other type inference techniques (RQ2). Black bars (sums) show the number of keys for which both techniques report types. Numbers are summation of all 10 packages.

more non-empty keys than other techniques, followed closely by Pytype. For the remaining packages where we only use default test suites, we have slightly worse coverage than Pytype on anaconda and invoke, reflecting worse test coverage by the underlying test suites. PoTo+ still has the highest average coverage percentage.

4.1.2 Equivalence (RQ2)

Next, we measure the similarity of inferred types. For each variable in a package, our analysis and the other type inference techniques can either have information on this variable, and the information is the set of its inferred types, or have no information, i.e., an empty set.

For built-in containers such as `dict`, `list`, `set`, and `tuple`, we compare only the top-level part and deem that they match if they are the same container types. The main reason for this shallow comparison is to facilitate processing and comparison as each type inference technique infers and reports types in different forms. For example, DLInfer reports only the type of a container, e.g., `dict`. Pytype reports parametric information, e.g., `Dict[str, int]`, but not always, while PoTo+'s inferred types come from abstract and concrete objects, e.g., `{'rename_handler': <class 'int'>}`. PoTo can collect parametric information, but it requires significant processing and raises issues on reporting types for polymorphic containers.

Matching is automatic but some comparisons require manual verification because of rendering of PoTo concrete objects. For example, PoTo+'s `1970-01-01` matches with DL-ST's `datetime.date` type. Another example is a match between PoTo+'s `<function <lambda> at 0x109297520>` and Pytype's `Callable[[Any],Any]`.

We are interested in 3 groups of equivalence: total match, partial match, and mismatch, where partial match is when the two sets of types have non-empty intersection and mismatch is an empty intersection. Figure 10 shows these as summation across the 10 Python packages. In addition, Figure 10 shows bars with the total number of non-empty keys by PoTo+ next to bars with the total number of non-empty keys by the other technique; for convenience it draws a bar that sums match, partial match and mismatch, i.e., the set of keys for which both techniques report a type. On all Python packages, PoTo+ shares the most non-empty keys with Pytype, and has high numbers of total matching. DL-ST and DL-ML have some

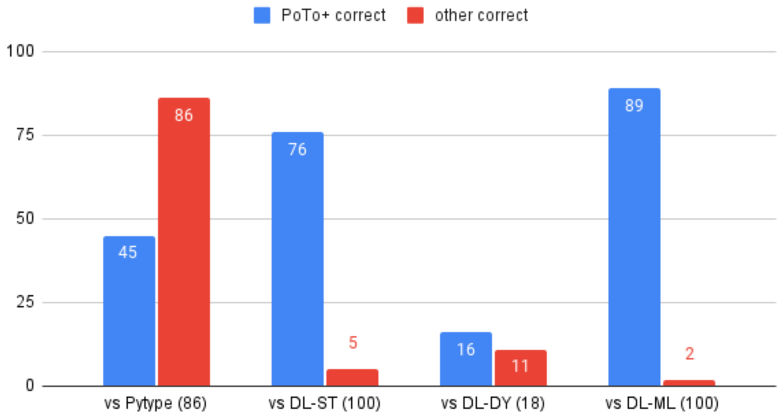

Figure 11 Correctness comparison between PoTo+ and other techniques (RQ3). Numbers in parentheses are # of samples. This figure zooms in on a sample of the mismatched keys from Figure 10; for Pytype there are 289 mismatched keys which is less than 1% of all variables.

matching and partial matching with our analysis, but also contain many keys that are mismatches. The discrepancy is discussed in Section 4.1.3. For the sc2 package, almost all common keys between PoTo+ and DL-ML are mismatches. The mismatches are due to simple assignments of class attributes in /sc2/ids/ directory, e.g. `NULL = 0`, `RADAR25 = 1`, `TAUNTB = 2`. Our analysis, Pytype, and DL-DY correctly infer the types as integer, but DL-ML labels all of them as `num` which is incorrect as it is not a type. There are 3,791 such assignments out of 3,984 common keys between PoTo+ and DL-ML in the sc2 package. Lastly, DL-DY has low coverage of types information, and most of them are a total match with our analysis.

Figure 10 shows that PoTo+ reports a larger number of types compared to the other techniques; for example, for Pytype, there are 29,252 keys for which both report type information (the black bar) and there are 13,016 keys for which PoTo+ reports a type but Pytype does not. In contrast there are 7,739 keys for which Pytype reports a type but PoTo+ is empty. Appendix A.1 includes per benchmark versions of this figure.

4.1.3 Correctness (RQ3)

To carry out the comparison we manually examine a sample of 10 non-match keys for each package and each pair of comparisons, i.e., PoTo+ vs. Pytype, PoTo+ vs. DL-ST, PoTo+ vs. DL-DY, and PoTo+ vs. DL-ML. In cases where there are fewer than 10 non-match keys, we exhaustively examine all of them, (e.g., there are only 8 non-matches between PoTo+ and Pytype for pygal, so there are 8 instead of 10 keys for pygal). There are only 18 total non-matches keys for PoTo+ vs. DL-DY. For the remaining 3 techniques, the total is slightly under 300 samples. Results are shown in Figure 11.

4.1.3.1 PoTo+ vs. Pytype

Comparison with Pytype shows that a mismatch usually means that PoTo+ is incorrect while Pytype is correct. Of the 86 pairs we examined, PoTo+ is correct 45 times, while Pytype is correct 86 times, because Pytype handles a larger set of Python features precisely, while PoTo defaults to Other, thus polluting points-to sets. For example, PoTo does handle list and dictionary comprehensions precisely, but defaults to Other on set comprehensions. Consider the example: `rampPoints = {p for p in rampDict if rampDict[p]}`. Recall that handling of

Table 2 Running times for PoTo and Pytype (RQ4).

Package	PoTo	Pytype
cerberus	51s	4m22s
mtgjson	1m39s	15m39s
pygal	1m48s	1h00m17s
sc2	52s	40m29s
zsfp	11s	7m28s
anaconda	1m59s	1h21m40s
ansible	6h23m07s	6h14m59s
bokeh	1h41m16s	10h15m41s
invoke	1m32s	20m45s
wemake	3m25s	9m17s

Other propagates points-to sets of `p`, `rampDict` and `rampDict[p]` into `rampPoints`. This leads to PoTo+ reporting type `[dict, bool]` as `p`'s set is empty and `rampDict`'s set contains a single dictionary object of `Point` object keys, and bool values (it is straightforward to add handling of set comprehensions and other features to PoTo to improve matching).

For the majority of cases where both are correct, PoTo+ reports a concrete type while Pytype reports a type parameter. For example, PoTo+ reports correctly `[dict, None]`, i.e., an optional dictionary for the return type of `_normalize_purge_unknown` in the cerberus package; while Pytype reports type variable `[_T0]` and this is correct with respect to Pytype's type system which allows for polymorphic functions.

4.1.3.2 PoTo+ vs. DL-ML, DL-ST and DL-DY

Of the 100 pairs we examine for PoTo+ vs. DL-ML, PoTo+ is correct 89 times, while DL-ML is correct 2 times. We mark PoTo+ incorrect conservatively — essentially, when the result is partially correct but not sound and not complete, we mark it incorrect. One common source of incorrect result for DL-ML was the `num` types mentioned earlier. Another source is usage of the function name instead of the function's return type at call assignments. Consider the following example from package pygal: `background = lighten(’#e6e7e9’,7)`. DL-ML reports type `[lighten]` for `background`, while PoTo+ correctly reports `[str, tuple]`. Yet another source of incorrect results is over-reliance on `dict` types. For example, in the wemake package: `AnyFunctionDef=Union[ast.FunctionDef,ast.AsyncFunctionDef]`, PoTo+ correctly infers type `[typing.Union[...]]` for `AnyFunctionDef` (via concrete evaluation). DL-ML infers `[dict]`.

Of the 100 samples we examine for PoTo+ vs. DL-ST, PoTo+ is correct 84 time and DL-ST is correct 5 times. There are no `num` types, as they are correctly assigned `int`; however, we observe similar patterns of function name instead of return type, and over-reliance on `dict`.

There are only 18 total mismatches for PoTo+ vs. DL-DY; 8 from ansible and 10 from bokeh. PoTo+ is correct 16 times and DL-DY is correct 11 times. Most cases are from accessing a dictionary or calling a function such as `children = kwargs.get(’children’)` or `typ = type(obj)`. On 5 occasions, DL-DY infers types that appear wrong such as `type`.

4.1.4 Scalability of PoTo (RQ4)

Table 2 show the running times of PoTo (total time for thousands of entry functions) and of Pytype. We run on a commodity Mac book Pro with 2.4 GHz 8-Core Intel Core i9 and 32 GB of Memory (one of our development machines). The "+" phase (type inference) is instantaneous and type aggregation and processing are negligible. We do not include these in the timing. In both cases, execution is IO-dominated, as PoTo writes points-to results into pkl files and Pytype's reveal-type reveals types by printing special error message.

PoTo runs in 1–3 minutes for all but the two largest packages, bokeh and ansible. Pytype is significantly more expensive ranging between 4 and 81 minutes on those packages. Without the reveal-type instrumentation Pytype runs 30% faster than with; thus, its underlying static analysis is expensive and PoTo still outperforms Pytype. PoTo runs slower than Pytype on ansible because ansible has nearly 500 test file (we filtered out ones that do not reference ansible packages from the 961 original ones to speed up testing) and many hit the same bottleneck of ansible code in the points-to analysis.

4.2 Call Graphs

To strengthen the case for applicability of PoTo, we evaluate it on another classical client of points-to analysis: call graph construction. We call the client PoToCG. We compare PoToCG to PyCG, the leading Python call graph construction analysis [34]. Unfortunately, PoToCG and PyCG are two different analyses, with different goals, and comparing them is non-trivial. PoToCG is a standard whole-program reachability analysis [3, 33, 23], whose goal is as follows: given a *main function* and a package, compute the smallest, i.e., most precise, call graph of reachable functions. PyCG, on the other hand, takes a set of files as input and computes a call graph that aims to include every call site in these files. Both graphs are unsound, by virtue of being analyses of Python, and both techniques aim to minimize the number of missing call graph nodes and edges. Nevertheless, PyCG is still the best Python call-graph analysis to compare to representing the state-of-the-art, and we made the comparison as apples-to-apples as possible modulo the different nature of the two approaches. We do not expand PoToCG to unreachable functions for this comparison, because such an expansion would entail a brand new call graph analysis that is antithetical to the reachability analysis inherent in Andersen-based call graph construction. Instead, we report on the number of reachable functions in PoToCG compared to PyCG, then we zoom in on the PoToCG-reachable functions and compare the quality of the PoToCG call graph for those functions compared to PyCG.

The research questions we address are as follows:

RQ5: How does PoToCG compare to PyCG in terms of reachable functions?

RQ6: For the PoToCG-reachable functions, how *complete* and how *precise* is the PoToCG call graph compared to PyCG?

We construct the PoToCG call graph by traversing call 3-address statements $\mathsf{x} = \mathsf{y}(\mathsf{z})$ in PoTo reachable functions and examining the points-to set of y. Meta-func objects represent abstract functions and lead to callees in the package. Concrete objects lead to callees that are either built-in functions (e.g., `len`) or external library functions (e.g., `re.sub`).

Consider function `UnconcernedValidator.__init__` in benchmark cerberus containing the following statement: `self.document_error_tree = errors.DocumentErrorTree()`. It gives rise to a corresponding 3-address-code call statement $\mathsf{x} = \mathsf{y}(\mathsf{z})$. Analysis variable y's points-to set contains meta-func abstract object `cerberus.errors.DocumentErrorTree.__init__`. Thus, we record a call graph edge from caller `UnconcernedValidator.__init__` to callee function `cerberus.errors.DocumentErrorTree.__init__` (this edge is missing from the PyCG results). PoToCG handles calls to built-in functions as well by recording the concrete object corresponding to the built-in function, e.g., `len`, `int`, etc. PyCG captures built-in callees as well. We aggregate results across all test cases, as in the type inference client.

Unfortunately, PyCG ran only on the five smaller packages — cerberus, pygal, mtgjson, sc2, and invoke. It did not terminate on any of the remaining packages. (PoToCG terminates on them.) To run PyCG, all files in a package are passed to PyCG as entry points. The five packages that terminated took 30 seconds to a few minutes, which are about the same as PoToCG as shown in Table 2. The other packages timed out after two hours per package.

Table 3 Function coverage of PoToCG and PyCG (RQ5). * indicates that the number is unusual due to non-existent callees, i.e., callees that do not correspond to function definitions.

FUNCTIONS	cerberus	pygal	mtgjson	sc2	invoke
Total functions	200	354	237	567	782
Funcs without a call stmt	38	83	32	256	111
Funcs with call stmt	162	271	205	311	671
Funcs in PoToCG	91	218	211	266	99
Funcs in PyCG	161	816*	259	412	738
Funcs in PoToCG but not in PyCG	5	26	0	15	2
Funcs in PyCG but not in PoToCG	75	624	48	161	641
Funcs in both. Same # of edges	37	88	86	168	50
Exactly the same set of edges	29	71	57	123	37
Different set of edges	8	17	29	45	13
Funcs in both. #Edges: PoToCG > PyCG	43	54	76	67	33
Funcs in both. #Edges: PyCG > PoToCG	6	50	49	16	14

Table 4 Edges in PoToCG reachable functions (RQ6).

EDGES	cerberus	pygal	mtgjson	sc2	invoke
True positive PoToCG	**339**	**599**	**679**	**438**	**260**
True positive PyCG	192	398	550	351	220
False positive PoToCG	**1**	**0**	**0**	**0**	**0**
False positive PyCG	12	813	116	4	6

Table 3 shows function coverage of PoToCG and PyCG (RQ5). The top three rows provide the raw statistic of each package we produced by scanning its Python AST. For the result of both techniques (row 4 and 5), we are interested in functions that have at least one outgoing edge or one incoming edge in their respective call graphs, meaning that we exclude isolated empty functions. Except for invoke, which relies entirely on its test suite (we did not write additional tests for it), PoToCG achieves relatively good coverage: 49% for cerberus, 72% for pygal, 91% for mtgjson, and 77% for sc2. We measure coverages by percentages of functions in PoToCG that have call statements, to total functions with call statements. PyCG numbers for pygal and mtgjson exceed the upper bound of total functions, because PyCG reports caller and callee nodes that cannot be mapped to actual function definitions, e.g., `pygal.etree.etree.fromstring.findall.node.serie.get.split.append`. PyCG includes each module as a node as it can potentially have call edges. PyCG also maps functions in __init__.py files differently. For example, function `safe_load()` in invoke/vendor/yaml3/__init__.py has its path as `invoke.vendor.yaml3.safe_load`. We adjust PoToCG to accommodate these cases.

Table 4 zooms in on the functions that are both in PoToCG and PyCG for which the two techniques report *different sets of callee functions* (RQ6). These are the bottom three lines in Table 3 surrounded with a solid rectangle. We examined all the edges in these functions. We classified a caller-callee edge as *true positive* when reaching a specific call statement in the caller would indisputably lead to the callee for some receiver (when there is a receiver); this is the case for the `DocumentErrorTree` constructor example above. We classified an edge as *false positive* otherwise. In practice, essentially all false positives were in PyCG where the reported callee could not be mapped to a function definition in the package or in an external library. One can see in Table 4 that PoToCG reports considerably better call graphs for the functions it reaches — it reports a much larger number of true positive callees compared to PyCG as well as virtually no false positives, while PyCG reports a large number of false positives for two of the benchmarks. Therefore, PoTo's call graph is more complete and more precise for the functions it reaches. We observed that when reported callees could be mapped to function definitions, both PoToCG and PyCG's callees were true positives. The reason why PyCG had a large number of false positives was that it reported many callees that did not correspond to function definitions.

We consider cases that highlight PoToCG's better recall (i.e., soundness) and better precision. PoToCG's better recall is due to its flow analysis as in the following example:

```
validator = self._get_child_validator(..); mapping[field] = validator.normalized(..)
```

In this example, the analysis must infer that a `Validator` object flows to the `validator` variable. PoTo infers that the analysis variable corresponding to `validator` points to a Validator object and correctly infers a call edge to `cerberus.base.UnconcernedValidator.normalized`. In contrast, PyCG leaves this call empty.

Another reason for PoToCG's better coverage is concrete evaluation. Consider

```
result = validator.normalized(...); mapping[field] = value_type(result.values())
```

In the above code, an empty concrete dictionary flows to the analysis variable corresponding to `result` and concrete evaluation deduces that the `values` member function is called. In contrast, PyCG does not infer a callee at this call.

As mentioned previously, PyCG reports hundreds of non-existent callees as in the following example: `pygal.util.decorate.serie.node`. Here, `decorate` is a function in the `util` module. We surmise PyCG infers `decorate` to return an object that has a field `serie`; however, it is not clear how that access path corresponds to an actual function definition in the pygal package. Another example is `pygal.graph.public.PublicApi.svg.serie.node`. We observed large numbers of such false positive callees in the pygal and mtgjson benchmarks but nearly none in the remaining benchmarks. In contrast, PoToCG reports were precise — if the call is reached then the reported function will be called.

Overall, we observed that recall (i.e., soundness) rather than precision is a more pressing issue for all analyses. PoToCG has better recall overall for the functions it reaches as detailed in Table 4, however, in the absence of ground truth it is difficult to judge how complete PoTo's call graph is. We did not observe precision issues in either analysis results, other than PyCG reporting non-existent callees. We further examined several applications and their hierarchies and we could not easily identify a call site that would have benefitted from a context- or object-sensitive analysis. For example, in pygal, a charting library and one of the larger applications in the suite, classes encapsulated predominantly string and numerical values, as opposed to complex objects; superclasses contained nearly all the functionality; and there were no setter and getter functions. As a specific case, one class hierarchy had root `class Style` which defined several class-level string and numerical fields representing colors and fonts. It also defined a function `get_colors` which formatted color strings to interact with CSS. Subclasses of `Style` added a small number of string or numerical class-level fields and defined no functions. Appendix A.2 gives an example where PoTo shows a clear advantage over an object-sensitive analysis. We conjecture that Python applications use class hierarchies differently from Java although further studies are needed. We consider PoTo a necessary baseline. Improved precision, possibly via context-sensitive, is a promising direction for future work in Python program analysis.

5 Threats to Validity

Section 4.1.3 discusses correctness comparison that depends on limited samples out of thousands of mismatches. To mitigate this, we collect all 18 mismatches against DL-DY, plus 10 samples per packages for all packages across 3 other techniques, resulting in total of 300 hand-labeled examples with broad coverage.

The analysis relies on unit tests for entry points, thus presuming the existence of test suites and good test coverage of the test suites. To increase coverage, we added custom entry-points for 5 of the packages; notably, coverage remains robust even for those large packages where we used only the existing test suites.

Lastly, PoTo is a hybrid analysis that alternates between concrete and abstract evaluation, and it employs semantic choices to deal with Python's complexity. This makes the analysis unsound by design, but it is the typical kind of trade-off made by most real-world static analysis [24]. Our results with two client analyses show that PoTo compares favorably against respective state-of-the-art techniques, thus validating its trade-offs.

6 Related Work

This section discusses prior work related to each of the three contributions stated in Section 1.

Points-to analysis. At its core, PoTo is an implementation of Andersen's points-to analysis [2], but unlike the original, it works for Python and is hybridized with concrete evaluation. The only other points-to analysis for Python we have found is in Scalpel [22]; however, Scalpel's analysis is not based on Andersen's, does not use concrete evaluation, and the paper lacks empirical results. Few static analyses have been shown to work on real-world Python programs, including PyCG [34] (which finds call graphs) and Tree-sitter [8] (which is limited to syntactic queries) [8]. Neither PyCG nor Tree-sitter does points-to analysis, nor do they use concrete evaluation. This paper empirically compares PoToCG against PyCG.

Outside of Python, work on points-to analysis and related alias and value-flow analysis dates decades back. Chatterjee et al. [6] describe a context-sensitive points-to analysis for C++, Rountev et al. [33] present an Andersen-style analysis for Java, and Fähndrich et al. [10] define a context-sensitive points-to analysis for C, among many other works. Siu and Xue [38] and Shi et al. [35] present scalable value-flow analysis over LLVM IR. Jang and Kwang [18] define the first Andersen-like points-to analysis for JavaScript and Sridharan et al. [37] improve Andersen's analysis with better reasoning about property accesses. These analyses work over established 3-address code IR in LLVM, Wala, and Soot. In contrast, we needed to construct 3-address code for points-to analysis from Python AST constructs.

Recent works on points-to analysis for Java (e.g., [25], [19]) focus on context- and object-sensitivity to improve precision. Our analysis is a context-insensitive and flow-insensitive baseline; we envision that the 3-address code and hybridization will serve as a foundation for building context- and flow-sensitive analysis for Python in the future.

Hybridization weaving concrete and abstract evaluation. PoTo uses concrete evaluation to solve the problem of analyzing Python programs that use external libraries. Two other works hybridize abstract (i.e. static) with concrete (i.e. dynamic) analysis for Python: PyCT [7] (which does concolic testing) and Rak-amnouykit et al.'s analysis [32] (which finds weakest preconditions). Neither is based on points-to analysis, nor has been used for type inference.

Looking back outside of Python, the idea of combining static and dynamic analyses has a long history in program analysis of Java and JavaScript. Hirzel et al. [16] hybridize Andersen's analysis with dynamic analysis to handle Java's dynamic class loading. Grech et al. [13] define a hybrid analysis for Java that takes whole heap snapshots and integrates them into the analysis. Wei and Ryder [43] present blended analysis for JavaScript that collects traces of execution then uses static analysis to reason about flow of tainted values. Tripp et al.'s analysis for JavaScript [40] addresses the problem of modeling the DOM; it runs a dynamic analysis to concretize the external environment, followed by a static analysis in the concrete environment. Similarly, Laursen et al. [21] do a dynamic pre-analysis followed by a static analysis that uses runtime results; they target property access in JavaScript, a notorious issue in JavaScript codes [5]. Our analysis differs from all these works as it targets Python. It presents a novel blend of static and dynamic interpretation — it weaves concrete evaluation in both 3-address code generation and constraint resolution and elevates concrete objects to first-class status in the analysis. Park et al. [28]'s analysis of JavaScript does concrete execution of selected functions using carefully constructed objects; this is similar to our usage, however, their goal is to improve precision, while our goal is to improve soundness.

We view Toman and Grossman's Concerto [39] as most closely related to our work. Concerto is a combined concrete and abstract interpretation framework. It considers concrete state (for framework code) and abstract state (for application code) and allows calls from one domain into the other. Concerto is a theoretical framework, while we focus on Python and develop 3-address code generation and a specific analysis, Andersen's points-to analysis.

Type inference and call graph analysis for Python. PYInfer [9] and DeepTyper [15] use deep learning for Python type inference, and several other works combine deep learning with simple static analysis [1, 27, 29, 30, 44, 46]. This paper empirically compares PoTo+ against the latest of those, DLInfer [46], chosen because it compares with several earlier works and its artifact [45] has points-to sets for 10 real-world Python programs. A handful of other works use static analysis for Python type inference [11, 12, 14, 26]. Unlike PoTo+, none of these use points-to analysis nor concrete evaluation. In addition to DLInfer, this paper also empirically compares PoTo+ against Pytype [12], which is based on static analysis. We chose Pytype as a second baseline because it is widely used in practice and, like PoTo+, handles real-world Python programs [31]. Even though call-graph analysis is a popular client for points-to analysis in other languages, for Python, it has received little attention from the academic community. The main work is PyCG [34], and thus, we chose it as a baseline.

Instead of hybridizing static analysis with concrete evaluation, several works hybridize static analysis with machine-learning (ML). Xu et al. present a static type inference for Python augmented with ML for guessing types based on variable names [44]. Typilus uses static analysis to build a Python program dependency graph, then uses a graph neural network for type inference [1]. TypeWriter performs deep-learning (DL) type inference, then uses a static type checker to repair hallucinations [30]. Type4Py [27], HiTyper [29], and DLInfer [46] are primarily DL type inferences for Python, assisted by simple static analyses.

7 Conclusions

This paper presents PoTo, the first Andersen-style points-to analysis for Python. PoTo works on real-world Python programs, which use dynamic features as well as external packages for which source code is often missing (and which often involve non-Python code). To handle external packages, PoTo introduces a novel hybridization of Andersen's static analysis with dynamic concrete evaluation. In terms of clients, this paper presents PoTo+, a type inference built upon PoTo. While several recent papers explore deep learning for Python type inference, our results indicate that (at least as of now) static analysis solves this problem with superior coverage and correctness. This paper also presents PoToCG, a call-graph analysis built upon PoTo that performs better than the state-of-the-art for Python.

References

1 Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 91–105, 2020. doi:10.1145/3385412.3385997.

2 Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994. DIKU report 94/19. URL: ftp://ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z.

3 David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1996, San Jose, California, USA, October 6-10, 1996*, pages 324–341. ACM, 1996. doi:10.1145/236337.236371.

4 Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, page 69–82, 1996. doi:10.1145/236337.236343.

5 Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:28, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2022.3.

6 Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, page 133–146, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/292540.292554.

7 Yu-Fang Chen, Wei-Lun Tsai, Wei-Cheng Wu, Di-De Yen, and Fang Yu. Pyct: A python concolic tester. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 38–46, 2021. doi:10.1007/978-3-030-89051-3_3.

8 Timothy Clem and Patrick Thomson. Static analysis at GitHub. *Communications of the ACM (CACM)*, 65(2):44–51, jan 2022. doi:10.1145/3486594.

9 Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang. PYInfer: Deep learning semantic type inference for python variables, jun 2021. URL: https://arxiv.org/abs/2106.14316, doi:10.48550/arXiv.2106.14316.

10 Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 253–263, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/349299.349332.

11 Levin Fritz and Jurriaan Hage. Cost versus precision for approximate typing for python. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 89–98, jan 2017. doi:10.1145/3018882.3018888.

12 Google. pytype, jan 2016. Retrieved June, 2024. URL: https://github.com/google/pytype.

13 Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: countering unsoundness with heap snapshots. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi:10.1145/3133892.

14 Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. MaxSMT-based type inference for Python 3. In *Conference on Computer Aided Verification (CAV)*, pages 12–19, 2018. doi:10.1007/978-3-319-96142-2_2.

15 Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 152–162, 2018. doi:10.1145/3236024.3236051.

16 Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. In *European Conference for Object-Oriented Programming (ECOOP)*, pages 96–122, 2004. doi:10.1007/978-3-540-24851-4_5.

17 IBM. T. j. watson libraries for analysis (wala), 2006. Retrieved June, 2024. URL: https://github.com/wala/WALA.

18 Dongseok Jang and Kwang-Moo Choe. Points-to analysis for javascript. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 1930–1937. ACM, 2009. doi:10.1145/1529282.1529711.

19 Minseok Jeon and Hakjoo Oh. Return of cfa: call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022. doi:10.1145/3498720.

20 C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO)*, pages 75–86, mar 2004. doi:10.1109/CGO.2004.1281665.

21 Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. Reducing static analysis unsoundness with approximate interpretation. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1165–1188, 2024. doi:10.1145/3656424.

22 Li Li, Jiawei Wang, and Haowei Quan. Scalpel: The Python static analysis framework, feb 2022. URL: https://arxiv.org/abs/2202.11840, doi:10.48550/arXiv.2202.11840.

23 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.*, 2(OOPSLA):141:1–141:29, 2018. doi:10.1145/3276511.

24 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM (CACM)*, 58(2):44–46, jan 2015. doi:10.1145/2644805.

25 Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi:10.1145/3360574.

26 Eva Maia, Nelma Moreira, and Rogério Reis. A static type inference for Python. In *Workshop on Dynamic Languages and Applications (DYLA)*, 2011. URL: http://scg.unibe.ch/download/dyla/2011/dyla11_submission_3.pdf.

27 Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4Py: Practical deep similarity learning-based type inference for Python. In *International Conference on Software Engineering (ICSE)*, pages 2241–2252, may 2022. doi:10.1145/3510003.3510124.

28 Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. Accelerating javascript static analysis via dynamic shortcuts. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1129–1140, 2021. doi:10.1145/3468264.3468556.

29 Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: a hybrid type inference approach for Python. In *International Conference on Software Engineering (ICSE)*, page 2019–2030, may 2022. doi:10.1145/3510003.3510038.

30 Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. TypeWriter: Neural type prediction with search-based validation. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 209–220, nov 2020. doi:10.1145/3368089.3409715.

31 Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. Python 3 types in the wild: A tale of two type systems. In *Dynamic Languages Symposium (DLS)*, pages 57–70, 2020. doi:10.1145/3426422.3426981.

32 Ingkarat Rak-amnouykit, Ana Milanova, Guillaume Baudart, Martin Hirzel, and Julian Dolby. Principled and practical static analysis for Python: Weakest precondition inference of

hyperparameter constraints. *Software – Practice and Experience (SP&E)*, 54(3):363–393, mar 2024. URL: `http://doi.org/10.1002/spe.3279`, `doi:10.1002/spe.3279`.

33 Atanas Rountev, Ana L. Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In Linda M. Northrop and John M. Vlissides, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001*, pages 43–55. ACM, 2001. `doi:10.1145/504282.504286`.

34 Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python. In *International Conference on Software Engineering (ICSE)*, pages 1646–1657, may 2021. `doi:10.1109/ICSE43902.2021.00146`.

35 Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: fast and precise sparse value flow analysis for million lines of code. *SIGPLAN Not.*, 53(4):693–706, jun 2018. `doi:10.1145/3192366.3192418`.

36 Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In *Datalog Reloaded*, 2011. URL: `https://doi.org/10.1007/978-3-642-24206-9_14`, `doi:10.1007/978-3-642-24206-9_14`.

37 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 435–458. Springer, 2012. `doi:10.1007/978-3-642-31057-7_20`.

38 Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 265–266. ACM, 2016. `doi:10.1145/2892208.2892235`.

39 John Toman and Dan Grossman. Concerto: a framework for combined concrete and abstract interpretation. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. `doi:10.1145/3290356`.

40 Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 49–59, 2014. `doi:10.1145/2610384.2610385`.

41 Raja Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, Montreal, oct 2000. URL: `https://dl.acm.org/citation.cfm?id=782008`.

42 Yin Wang. Pysonar2 - a semantic indexer for python with interprocedural type inference, dec 2017. URL: `https://github.com/yinwang0/pysonar2`.

43 Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 336–346, 2013. `doi:10.1145/2483760.2483788`.

44 Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 607–618, 2016. `doi:10.1145/2950290.2950343`.

45 Yanyan Yan. DLInfer dataset, jan 2023. `doi:10.5281/zenodo.7575545`.

46 Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu. DLInfer: Deep learning with static slicing for python type inference. In *International Conference on Software Engineering (ICSE)*, pages 2009–2021, may 2023. `doi:10.1109/ICSE48619.2023.00170`.

47 Yi Yang, Ana L. Milanova, and Martin Hirzel. Complex Python features in the wild. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, pages 282–293. ACM, may 2022. `doi:10.1145/3524842.3528467`.

A Appendix

A.1 Per-application Equivalence Comparisons including Bars for Non-Empty Variables

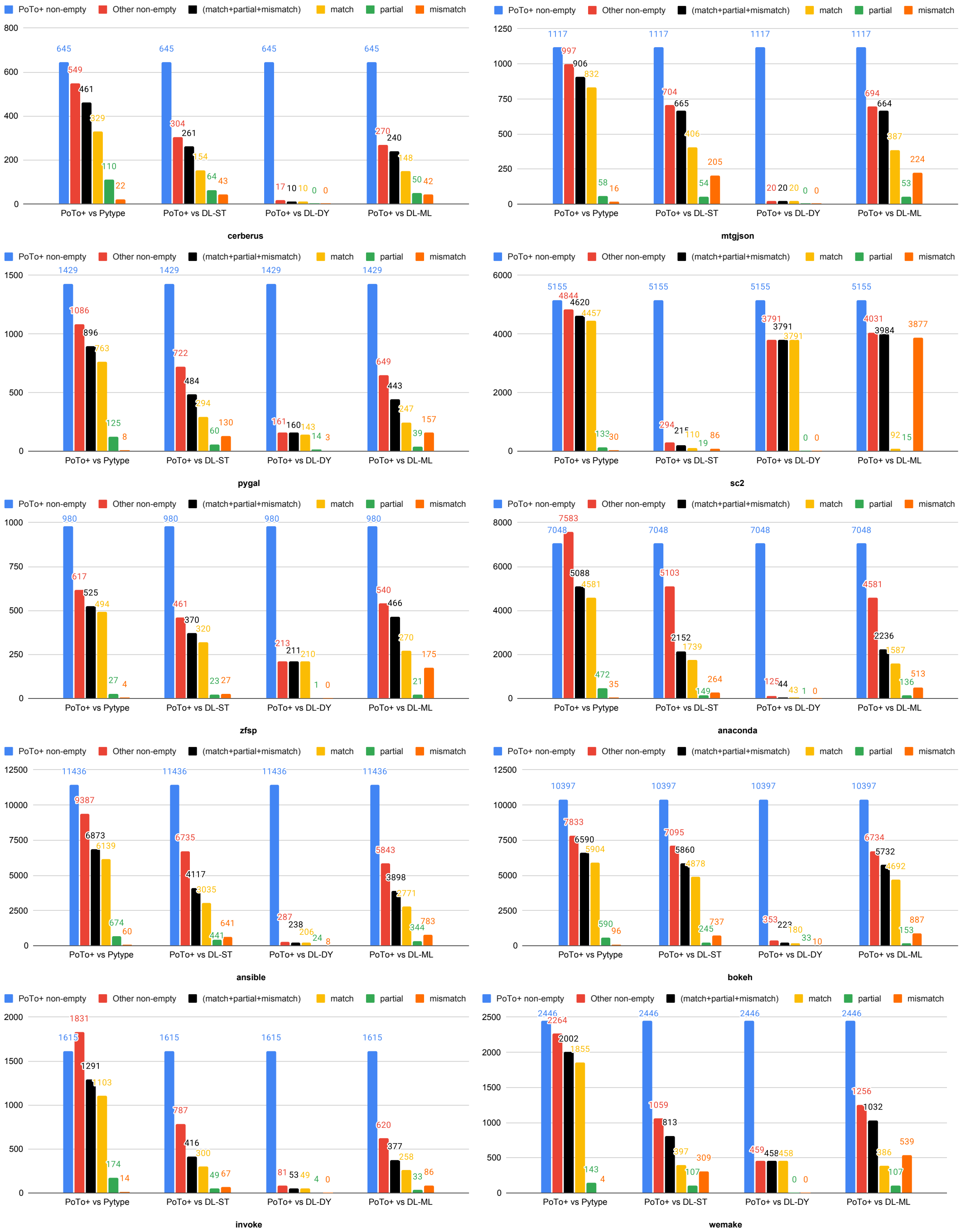

Figure 12 Equivalence comparison between PoTo+ and other type inference techniques (RQ2) for each individual package

A.2 Difficult-to-Analyze Call Sites

```
 1 class PublicApi:
 2    ...
 3    def render(self):
 4      self.setup()
 5      self.svg.render()
 6
 7 class BaseGraph(PublicApi):
 8    ...
 9    def setup(self):
10      self.svg = Svg(self)
11
12 class Line(BaseGraph):
13    ...
14    # Defines several new functions
15    # Does not override render or setup
16
17 # A test:
18 line1 = Line()
19 line1.show_legend = False
20 line1.fill = True
21 line1.pretty_print = True
22 line1.no_prefix = True
23 line1.x_labels = [’a’, ’b’, ’c’]
24 line1.add(’_’, [1, 2, 3])
25 line1.render()
26
27 line2 = Line(show_legend=False, fill=True, pretty_print=True, no_prefix=True,
       x_labels=[’a’, ’b’, ’c’])
28 line2.add(’_’, [1, 2, 3])
29 line2.render()
```

The listing shows an excerpt of a hierarchy from pygal. The majority of functionality is in base classes `PublicApi` and `BaseGraph` and the initialization context for graphing objects (e.g., line) consists largely of boolean, string, and numerical values. In this example the two line objects are initialized with the same values, as the test case tests their equivalence, but the important point is the types of encapsulated objects (booleans and lists of strings or integers), not the values themselves.

The call at line 5 illustrates a performance advantage of PoTo over a 1-object-sensitive analysis. Clearly, PoTo resolves the call to `Svg.render()`. A 1-object-sensitive points-to analysis will replicate functions `render` and `setup` per each one of the line objects hoping to distinguish between the two initialization contexts. However, replication is redundant as field `svg` is initialized in each replica of `setup` to the same `Svg` object, which flows into each replica of `render`; the analysis therefore discovers the same call target twice. While this is one example and we do need further studies, we observed similar flows in pygal and other applications — initialization context is of values of simple types that have no impact on the flow of objects and no impact on the call graph.