FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Václav Luňák

Extending Data Lineage Analysis for Python with Runtime Types

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	doc. RNDr. Pavel Parízek, Ph.D.
Study programme:	Computer Science
Study branch:	Software and Data Engineering

Prague 2024

I would like to thank my supervisor, doc. RNDr. Pavel Parízek, Ph.D., whose expertise and helpful suggestions were invaluable in making this work possible.

For all their help along the way, I want to also thank my colleagues at Manta, and especially my teammates, who were always willing and able to assist me with whatever I needed.

My undying gratitude goes to my parents, who have been supporting me throughout my studies with their love, support, and food. I wouldn't be here without you.

Title: Extending Data Lineage Analysis for Python with Runtime Types

Author: Václav Luňák

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: There is an increasing demand in the domain of data science for automated tools analyzing the data lineage of software systems. In situations where general-purpose programming languages are used, Python is among the most popular choices. It is also one of the most challenging to analyze. Manta Flow is an automated data lineage analysis platform that contains a scanner for Python. In this work, we developed an extension of this scanner. Its purpose is to statically determine the types of expressions in an analyzed application. We achieved this by expanding the concept of data flows to carry type information and we appropriately refactored the internals of the scanner. This information was then used to implement an improved method for finding the targets of function invocations during the analysis of data flows.

Keywords: Python, data flow, data lineage, type inference, Manta

Contents

Chapter 1

Introduction

Modern software is becoming increasingly complex. We shall make no effort to quantify this proposition, nor do we wish to fully examine the reasons for why it is true. Nevertheless, as unsubstantiated as our claim is, it seems fairly uncontroversial to suggest that on average, the complexity of software has been increasing as a function of time.

This phenomenon, of course, is not new. As early as the nineteen sixties, computer scientists had to contend with what was at the time dubbed the *software crisis* [1], as the increasing power of new hardware and correspondingly increasing demands made of said hardware outpaced the programmers' ability to efficiently develop new software. In time, various software engineering techniques were developed that aimed to address these issues. Still, the theoretical capabilities of computers have kept growing, and so have the expectations we have of software today. We can but wonder what the computer scientists of half a century ago would have to say about the beginner web developer of today who struggles to even start working on their ”Hello world” website because their Typescript compiler has suddenly become incompatible with their version of Webpack and one of their six code analysis tools does not support ESM modules, which is coincidentally the only way a dependency of a dependency can be imported.[1]

Perhaps nowhere is this complexity more readily apparent than in the realm of enterprise software. Large corporations now have the resources and inclination to tackle unimaginably sophisticated problems, combining the efforts of thousands of developers to do so. On such scales, any one person may have a firm grasp on the inner workings of one small part of the system and, given a sufficient level of abstraction, may have some understanding of its overall structure, but singular intimate knowledge of every single part—not to mention how it interacts with all the other parts—becomes impossible to maintain.

[1]Based loosely on conversations between the author and several aspiring web developers, as well as his own experiences.

Unable to rely purely on human knowledge, we once again turn to software for assistance. Using automated code analysis tools, a stakeholder can examine even large projects for detailed insights that would be nearly impossible to obtain otherwise. One domain of such examination that is of particular interest to us is *data lineage*, that is, the process of tracking and visualizing where data originates, how it changes, and where it ultimately ends up in the data pipeline. Whether it is for assessing regulatory compliance, performing impact analysis, or simply trying to gain a more complete understanding of some aggregated piece of data, data lineage is becoming a useful tool for businesses with sizeable software stacks.

Manta, with its automated data lineage platform Manta Flow, has become one of the key players in this space. It is able to scan more than fifty different technologies, encompassing a broad suite of databases, reporting tools, modeling platforms, and data integration tools. However, in this work, we focus on one scanner in particular, which is the Python scanner.

In the field of data science—and to some extent also in general-purpose programming—Python has become a popular language of choice, providing some interface for most commonly used frameworks and technologies. In addition, tools used for data transformations have begun introducing the ability to define those transformations using general-purpose programming languages. As a result, there is a demand for tools that are able to generate lineage information for source code written in Python. After all, trying to track the data lineage of a process will not be very fruitful unless you are able to analyze the whole process. If there's an errant script in the middle of the pipeline that you cannot scan, what happens after it is anybody's guess.

Manta responded to this demand by adding a Python scanner to its platform. This development started as a student software project in 2021 and has been continuing ever since. However, Python is by its nature a very dynamic language, which, as one might expect, makes it challenging to analyze.

For example, let us consider such a fundamental concept as function invocation. When we encounter an expression in the form of `x.foo()`, which function is actually being called? In languages with a more static type system, we can make a fairly accurate prediction based on the type of the expression `x`, possibly allowing for minor variance with constructs such as the virtual method table. Python gives us no such luxury. It always determines the invocation target dynamically, at runtime, depending on the exact type of the object stored in `x` at the time of the call. As we are working with a dynamic language, the variable gives absolutely no indication of which types it may contain. For any static analysis tool working with Python, this makes such determinations difficult at the best of times, and impossible in general. The example in listing Listing 1.1 illustrates just one of the difficulties a static analysis tool may encounter.

Listing 1.1 Statically Undecidable Invocation

```python
from module import A, B

if input() == "a":
    x = A()
else:
    x = B()

# A and B may be fully disjoint types
# They just both contain a method named foo
x.foo()
```

1.1 Goals

Due to the challenges associated with analyzing Python code discussed above, the current implementation of the Python scanner in Manta Flow suffers from a lack of accuracy when trying to resolve the targets of function invocations. This leads to over-approximation and therefore reduced overall usefulness of the results it produces. One way to address those problems is to imbue the scanner with more information about the types of objects that can be stored within variables, based on the idea that the more we know about those types, the more precise we can be in determining which functions could be called. The main goal of this thesis is to find a way to track this type information throughout the analyzed program. The potential ways this information could then be used are many. This thesis focuses on one of them, namely designing and implementing a better way of resolving function invocation targets.

1.2 Outline

Before we can describe the proposed improvement in more detail, we need to gain some fundamental understanding of the current state of affairs and its limitations. Chapter 2 gives a brief overview of the Manta Flow platform as a whole and outlines the structure and the main components of the Python scanner, and Chapter 3 expands in more detail on the issues that this work aims to address. Chapter 4 contains a detailed analysis of areas important to this work. In particular, it focuses on Python's type system, its rules for function invocation, and known approaches used for type inference. Chapter 5 builds on this analysis and describes the overall design used in this thesis and describes the type analysis algorithm chosen for this work, *object origin tracking*. Chapters 6 to 9 each describe the design and implementation of one part of this work, namely the algorithm for tracking runtime types, the extension of the scanner's plugin system to accommodate type analysis, the creation of a new call graph, and integration into the scanner. Chapter 10 then provides a detailed evaluation of the impact this work has on the scanner in terms of both performance and precision.

Chapter 2

Manta Flow Overview

The Manta Flow platform has been the subject of several theses and student projects in the past. As such, it has already been extensively covered by others [2, 3, 4] and its various intricacies are unimportant to this work. Nevertheless, it would be remiss of us not to give at least a brief outlook on its design and structure, which we do in the first section of this chapter. The rest of the chapter then explains in more detail the specifics of the Python scanner, with a special focus on those components with which this work has to interact in some fashion.

2.1 Manta Flow

Manta Flow is a platform for automated analysis of data lineage. Given a software system, it produces a data lineage flow graph, which shows how data can travel throughout that system. Using Manta Flow, customers can simplify data governance tasks, lower the time necessary to resolve data incidents, and overall gain better insight into their data pipelines. Where a manual analysis of a large interconnected system could take several people weeks to complete, the automated nature of Manta Flow means that, even in enterprise-scale deployments, it can produce results in at most a small number of days, and possibly even in a matter of hours.

Instead of generating the lineage dynamically by observing the system as it runs—an approach taken by some data lineage tools—Manta Flow uses metadata for its analysis. In a database, for example, the actual values stored are irrelevant, it is only interested in the schema; for a business intelligence tool, it does not need any generated reports, just the scripts used to generate them, and so on. The upside of this strategy is that the platform doesn't have to rely on an event happening in order to notice it. A dependency between two services can be found even if it was not utilized during the time the system was observed. The obvious challenge is that this approach has to rely on statically analyzing the metadata, which is overall more difficult and, depending on the analyzed technology, can lead to over-approximations, finding some connections that are not realistically possible.

Because the platform supports many different technologies, it has to allow for many different strategies for extracting and analyzing said metadata. A PostgreSQL database cannot be processed the same way as a PowerBI script or a Kafka schema. This problem lends itself to a modular solution, and Manta Flow indeed has a highly modular design. Each of the more than forty scanners that are currently implemented uses the same basic interface, which consists of two parts:

1. The *connector*, further subdivided into two components:
    (a) The *extractor*, which gathers all the necessary inputs from the analyzed software and transfers it into a central well-defined location.
    (b) The *reader*, which then processes those inputs to create a general model of the analyzed application.
2. The *dataflow generator*, which takes the output of the reader and uses it to generate a data lineage graph.

Once all the scanners finish their work, the resulting individual graphs are then centrally combined into the final interconnected graph, which is presented to the user. This design allows each scanner to handle the particularities of its own technology while providing a common abstraction that is able to interconnect anything from databases to general-purpose programming languages.

2.2 Python Scanner

As mentioned in the previous section, each scanner in Manta Flow consists of two main parts—the connector and the dataflow generator. The dataflow generator processes the results of the connector to generate the actual lineage graph. It is not necessary to delve into the details of its operation for Python, as they are unimportant to our work. The connector is further subdivided into two components—the *extractor* and the *reader*. The work of those components can be described in five individual steps (see Figure 2.1).

2.2.1 Extraction

In every scanner, the extractor is responsible for gathering data relevant for the analysis and transferring it to the machine running Manta, in a format and

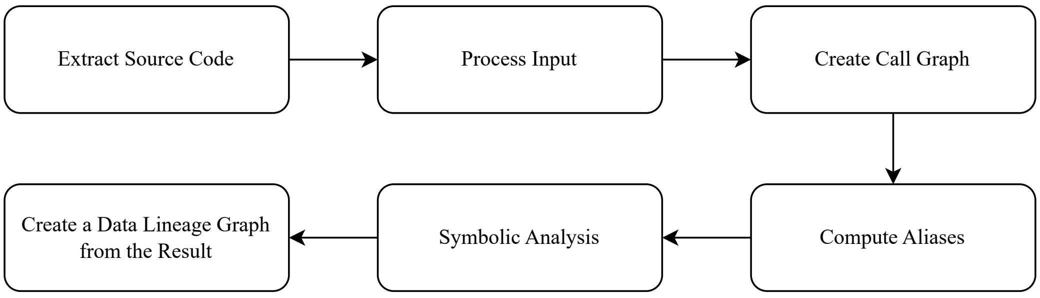


Figure 2.1 Steps Performed by the Python Scanner

location that the *reader* can work with. For the Python scanner, this essentially means collecting the source code of all analyzed files. The main focus is on extracting application code, that is, code written by the user. To provide the most accurate lineage possible though, it may be also necessary to extract some *library code*, i.e. code from another package—not written by the client—that is used from the application.

The *reader* is designed to work directly with Python code, so the extractor does not need to perform any transformations on the files it gathers. Simply transferring them to a predetermined location is enough.

2.2.2 Input Processing

The syntax of Python is full of syntactic sugar and constructs that may be generally useful for programmers but are of no importance to the *reader*. The first step performed by the *reader* is therefore transforming the code into a representation that is easier to work with. This bespoke processing tailored specifically to the *reader* allows us to make actually working with the code easier and gives us complete control over how it is presented to the rest of the scanner.

One illustrative example of such a transformation can be found in the so-called magic methods. In Python, virtually every operator can be replaced with a call to a specially named function. For example, the expression `'Hello' + 'world'` can be equivalently written as `'Hello'.__add__('world')`. Similarly, `list[x]` can be represented as `list.__getitem__(x)` and a `for` loop can be replaced by a call to `__iter__()` followed by successive calls to `__next__()`.

This is a feature that is mainly intended for operator overloading, but it is also very useful for our purposes here. By transforming each such operator expression into its equivalent method call, not only does the scanner not need to worry about overridden operators, it does not need to explicitly handle almost any operators in the first place. All that is necessary is the ability to handle called functions.

2.2.3 Call Graph Construction

The third step, internally named *infrastructure analysis*, is concerned with constructing the call graph. The call graph can be thought of as a directed graph that encapsulates which functions are called from any other function in the application. Determining which functions are reachable from any given scope can be difficult, so pre-computing this information can save a lot of time during analysis. (At least, that is the hope. The deficiencies of this process, as described in Chapter 3 are in fact one of the main motivators of this work.)

Infrastructure analysis works with executables. An executable is simply some part of the program that can be executed. Specifically, an executable is either a function, a class, or a module that exists somewhere within the application.

There are two steps to infrastructure analysis. The first is the creation of a *descriptor map*. Each executable in the analyzed application is assigned a descriptor. Stored within this descriptor are parent–child relationships between executables. If one executable is contained within another (say a function is defined within a class), the latter is marked as the parent of the former. Each descriptor also contains a map of `import` statements that appear within it, with links to the executables being imported. This is a necessary part of call graph construction, as functions are—unsurprisingly—often called from different modules. Being unable to traverse the module boundaries would amount to a significant loss of available information.

The second step is the actual construction of the call graph. The nodes of this graph are the descriptors created in the first step. The edges represent caller–callee relationships; an edge exists between two descriptors if the body of one contains an invocation of the other.

2.2.4 Alias Computation

Within a program, some variables may be aliasing the same object. Consider the following simple example:

```python
a = [1, 2]
b = a
b.append(3)
```

At the end of this snippet, `b` contains the value `[1, 2, 3]`, but so does `a`, as they are both aliasing the same underlying value. This is not necessarily obvious just from looking at the individual instructions, and it has to be accounted for in the analysis in order to correctly detect and propagate data flows.

As a precursor to the actual analysis, the *reader* visits assign statements and computes all aliases in different scopes of the analyzed application. Aliases are context-sensitive, so this computation doesn't store the actual values being aliased, but merely the equivalence classes between expressions in a given scope (i.e. “in function `foo`, variable `a` is an alias of variable `b`”).

2.2.5 Symbolic Analysis of Data Flows

Once all the prerequisite steps are completed, the actual symbolic data lineage analysis starts. The analysis being symbolic means that the code is analyzed statically, without any runtime information. This leads to some obvious losses in precision. For example, in any non-trivial `if/else` statement, it is impossible to statically determine which branch will be taken, meaning both possibilities have to always be considered.

The analysis starts from the entry point of the application, which is the executable (function or module) in which the application starts. It then proceeds using a mainstay of data flow analysis, a worklist algorithm.

Each entry that is placed inside the worklist is represented by an invocation context. An invocation context encompasses the function being called, the place it is being called from, and any flows associated with the provided arguments. When processing an executable, the analyzer computes an *executable summary*, which represents the complete data flow information for that particular invocation context. Whenever an executable summary changes, all the callers and all the callees of that executable must be (re-)added to the worklist, as their flows may be affected by this change as well. The analysis ends when the worklist becomes empty, which marks a fixed point and means no new flows are being generated.

Within the analysis of each invocation context, the analyzer maintains a list of tracked expressions and their associated flows. At the beginning of a function, this list includes the parameters of the function, and their associated flows are based on those of the provided arguments. Subsequent assignments and function calls inside the body of the executable may introduce new tracked expressions. For example, in an assignment expression, all flows associated with the expression on the right-hand side are propagated to the expression on the left-hand side.

After symbolic analysis concludes, its output is passed to the dataflow generator, which transforms it into the final lineage graph that can be visualized in Manta Flow.

2.2.6 Plugin-Handled Libraries

While introducing extraction, we touched on the need to handle library code as well as application code. However, one important way in which libraries differ from code provided by a customer is that library code is stable and its behavior is well-known in advance. The Python scanner takes advantage of this with its concept of plugins.

For specific often-used libraries (such as the standard library or Pyspark), the scanner never directly analyzes the code that defines the library implementation. Instead, for relevant functions in that library, an explicit *propagation mode* is defined, which emulates just the effect that function has on its inputs and outputs. When a call to such a function is found inside the application, it completely bypasses the worklist and just directly applies the configured propagation modes. This has massive benefits for both performance (as analysis of library code doesn't explode the worklist) and precision (as automatic analysis of library code would produce over-approximations and other faults which would then permeate the main application). On the other hand, these benefits are offset by the need to manually create and configure these propagation modes, so plugins are developed judiciously only for libraries that are actually used by customers.

Chapter 3

Limitations and Improvements

With the newfound understanding of how the Python scanner operates, we turn our focus to the issues that cause the aforementioned lack of precision and discuss the steps necessary for improvement.

3.1 Causes of Imprecision

The previous section makes note of infrastructure analysis and the call graph as one of the structures computed before the analysis begins. As it turns out, this concept has some major flaws. To explain those flaws, let us first dig a little deeper into what actually happens during infrastructure analysis.

3.1.1 Descriptor Map

The creation of the descriptor map works without issue. This need not be a surprise—after all, the structure of the descriptor map is influenced solely by factors that can be computed statically. Whether an executable is contained within another is apparent from the structure of the code and doesn't change during execution. What is imported by any given import statement is similarly straightforward to figure out. We therefore have all we need to construct a correct descriptor map without relying on any information produced at runtime.[1]

[1]An attentive reader may object that this statement is not completely accurate. Indeed, the dynamic nature of Python code grants it practically unlimited potential for modifying its own structure at runtime, including generating code and importing modules. We can name the `__import__()` and `setattr()` functions as representative examples. Usage of such mechanisms is however generally considered bad practice and is in fact so uncommon that the scanner is not required to support them, so they merit no further consideration.

3.1.2 Call Graph

Armed with the descriptor map, infrastructure analysis computes the call graph. In this graph, nodes are the executable descriptors, and an edge leads from one executable to another if the former contains an invocation of the latter.

There is the obvious problem of how these edges are computed. As we hinted at in the introduction, determining which invocation corresponds to which function in Python is difficult at the best of times, and impossible generally. The infrastructure analysis is therefore resigned to a fairly straightforward approach, which can be described in three steps:

1. Find the descriptor of the module in which the invocation was made.
2. From that module, recursively descending through imports and child descriptors, find all functions with a matching name.
3. For each function found, try to match the invocation arguments to the function parameters. If successful, include it in the result set.

This method is rooted in a general principle guiding the scanner, which is that considering impossible options is preferable to leaving out possible ones. It clearly doesn't miss any possibilities, and any actual executable that is invoked will always be found.

On the other hand, it is not hard to see why such a simplistic process results in wild over-approximation. For one, it doesn't take into account what the function is actually being called on, or if it's being called on anything at all. Static functions may be matched on what is a call of an instance method, and vice versa. The identifiers used in import statements are also completely ignored, gleefully passing through modules that couldn't possibly have been referenced by the invocation expression.

Listing 3.1, consisting of two modules, illustrates the issue. On line 20, we see the invocation `x.foo()`. Purely intuitively, we are able to determine that `A.foo` is the only possible target for that invocation, as `x` always holds an instance of `A`. The algorithm is unable to come to the same conclusion. It also considers `B.foo` as a possible target, despite it being on a class that's never instantiated. It even considers `module.foo` a possibility, as it is reachable from the `main` module through an import, even though the module isn't even referenced after being imported. There are some heuristics in place that prevent it from matching `main.foo`, as that function cannot be called on an object, but in practice, those don't help very much.

This example is admittedly contrived, but it shows a real issue with the scanner. In larger codebases, and especially in libraries, certain standard function names are omnipresent, which results in a highly connected call graph and an untenable

Listing 3.1 Ambiguous Function Call

```
 1  ## module.py ##
 2  def foo(x):
 3      print('module.foo')
 4
 5  ## main.py ##
 6  import module
 7
 8  class A:
 9      def foo(self):
10          print('A')
11
12  class B:
13      def foo(self):
14          print('B')
15
16  def foo(x):
17      print('foo')
18
19  x = A()
20  x.foo()
```

increase in superfluous generated data flows. This work aims to improve the way invocation targets are found, increasing precision to a point when any remaining over-approximations are generally acceptable and don't hinder the usefulness of the scanner.

3.2 Goals

Some problems with the algorithm are easier to solve. Returning to the example above, we can eliminate `foo` (line 16) as a possibility purely by considering the syntactic form of the invocation—line 20 calls a method on an object, so it clearly cannot end up resolving a global function. A similar thought process can eliminate `module.foo`.

Other improvements are harder. To decide that `B.foo` is not a possible target, we need to look at not only the invocation itself but the surrounding context as well. The only way to make sure of that is to make sure that `A` is the only type that is present in `x`. Our example makes that obvious, as the variable was assigned on the very previous line. In the general case, getting the necessary context is trickier. The variables may enter scope as function parameters, and they may arbitrarily change their value at runtime. What's more, the object of concern doesn't even have to be stored in a variable—any expression may produce an object on which we can directly call a method (`foo().bar().baz()`).

Behind this line of reasoning stands the core idea of this thesis. If we can accurately track the types of objects that are possibly stored in each expression, we can use those types to determine which functions are invoked on those expressions and improve the overall accuracy as a result.

With this more complete understanding of the problems with invocation target resolution, we can now specify that the desired improvement should occur in two ways:

- For calls to functions that are "static" (i.e. not instance methods), improve the search by taking into account more context surrounding the invocation.
- For calls to instance methods, create an algorithm that finds the types of those instances and uses them to find the targets.

In order to achieve those improvements, the thesis has the following concrete goals:

1. Analyze the mechanisms of function invocation in Python, its type system, and existing approaches used to analyze it.
2. Create a module that is able to parse the class hierarchy of Python applications.
3. Design and implement an algorithm for tracking runtime types of expressions in analyzed Python code.
4. Use the acquired data to create an improved method of finding targets of function invocations.
5. Integrate the above into the Python scanner and adjust the scanner based on the resulting improvements.

Besides these main goals, the thesis has one more slightly aspirational goal. The Python scanner is currently incapable of working with function objects that are assigned to variables or passed as callbacks to parameters of other functions. This deficiency is only tangentially related to the main goals of the thesis, in that it is a long-awaited area of improvement for the scanner which is somewhat related to locating function invocations. If we are able, we would like to implement support for this reassignment of function objects.

Chapter 4

Background Analysis

In this chapter, we look at some background information necessary for designing and implementing our extension to the scanner. First, since we are in the field of type analysis, we naturally look at the rules that govern the type system in Python. Then we analyze the ways functions are declared and invoked in a Python program. We also show some existing approaches to type inference that we considered and took inspiration from during this work.

As the first two sections are focused on various language features of Python, the information gathered within them was collected primarily from the official language documentation [5].

4.1 Python Type System

Python describes itself as an object-oriented language. As is the case with almost any modern language, the practicalities of writing code have eroded some of the purity attributed to any single programming paradigm, so this is not quite the whole story. If nothing else, Python is also well suited for a more procedural approach, to the point that someone opening a Python project could conceivably be greeted with a file containing just a series of plain functions, top-level statements, and some imports from other files written in a similar manner, giving an impression of a structure that seems more akin to C than Java.

Similarly, the apparent embrace of a dynamic program structure and disregard for any semblance of static typing may lead someone to think of the phrase "Python type system" as almost oxymoronic.

Despite these impressions, the language at its core is fundamentally object-oriented, and its type system is surprisingly robust, as we shall shortly demonstrate. If we want to be able to analyze how objects of different types propagate throughout the execution of a program, understanding these concepts is a necessary prerequisite. This section explains some key fundamentals in this area.

4.1.1 Data Model Basics

In Python, everything is an object. This includes all the data stored during execution as well as the executed code itself. The former is unsurprising, though we must point out that unlike in other object-oriented languages, there are no so-called primitive values—the number `1` is represented as an object like any other. The latter—representing parts of the program itself using objects—deserves some more attention.

The concept of "code as objects" is as old as object-oriented languages themselves, and is by no means unique to Python. Still, it is a notable point of differentiation from some other mainstream OOP languages. C++, for example, makes clear syntactic and semantic distinctions between objects and classes. In Java, programmers have access to class objects (those being instances of the `Class` class), but there are still restrictions imposed on classes that make them conceptually separate entities. In comparison, Python's approach is remarkably consistent.

If the interpreter finds a class definition, it interprets the body of that definition as a block statement and creates a new class object with the result. If a function is defined within that class, a function object is added as a property of the enclosing class object. Importing a module results in a module object being created, containing the variables defined within that module. This makes working with different kinds of objects homogeneous; the mechanism for accessing a function defined in an imported module is the same as getting a field of a class instance. It also means some constructs can be found in places one might not expect. It is entirely possible for a function to declare a class inside its body, return an instance of that class, and then effectively discard it once execution leaves its scope. Multiple invocations of such a function would produce objects that are each an instance of a completely disparate class, even if all the classes could effectively have the same name. Another common example of this is putting import statements inside `if/else` blocks, such that some modules are imported only under certain conditions, which can be determined completely at runtime[1]

4.1.2 Inheritance

As a class-based object-oriented language, Python of course has its own implementation of class inheritance. The topmost class in the entire class hierarchy is `object`, which all other classes inherit from, be it directly or indirectly. On the

[1]For example, big modules containing class definitions used only for type hints may be imported when a type checker is running, but omitted during "real" execution.

other side, there are single-valued bottom types, which are not acceptable as base classes. These are `None`, `Ellipsis`, and `NotImplemented`.

Each class can have one or more base classes. If a class defines no base classes explicitly, it is implicitly a direct descendant of `object`. Conversely, multiple base classes can be declared. This leads to obvious diamond problems—any two parents will have at least `object` as a common ancestor.

Such hierarchies present a challenge when invoking functions, especially when a `super()` call is used to explicitly invoke functions of base classes. When, for example, a class is first initialized, we want all the initializers defined on every superclass (and recursively, every superclass of those classes) to be executed so that the initial state of the object is configured as expected. On the other hand, none of these initializers should be run more than once for the same reason. Similarly, if multiple base classes define an override of the same method, how do we determine which one is called?

In languages with single class inheritance, the solution is trivial—simply walk through the chain of inheritance, starting with the instantiated class and moving up until the topmost point (whatever the language's equivalent of `object` is) is reached. For languages with multiple inheritance, where the ancestry of a class can look like any directed acyclic graph, that's not an option.

To address this, Python computes a *method resolution order* (MRO) of each class to determine how methods are overridden. An MRO results in a linear ordering of a class and all its ancestors, which can then be traversed much in the same way as in single class inheritance situations. This ordering is called a *linearization* of the class. Since version 2.3, it uses the C3 method [6] to generate MROs.

Let us consider a class C with base classes $C_1, C_2, ..., C_N$. The linearization of C, denoted $L(C)$, is calculated recursively as

$$L(C) = [C] + \text{merge}(L(C_1), L(C_2), ..., L(C_N), [C_1, C_2, ..., C_N]),$$

where $[C_1, C_2, ..., C_N]$ is a list of the enclosed elements and $+$ a concatenation of lists. The merge operation itself can be calculated as follows. Starting with the head[2] of the first argument, check whether it is in the tail of any other argument. If it isn't, remove it from all lists and add it to the result. Otherwise, check again with the head of the next argument. This process repeats until either all the arguments are empty, or no suitable head has been found, in which case linearization is impossible and an exception is raised.

Another quirk of Python's inheritance mechanism is that the base classes need not be known until the class is constructed. As previously mentioned, a

[2]In this explanation, the *head* is the first element in a list, and the *tail* is the list of all other elements.

class definition is merely a kind of statement that creates a class object. The list of base classes is therefore also only resolved once the class definition is interpreted. This means, among other things, that any expression can be used as a base class definition, as long as that expression results in a class object. Listing 4.1 shows these possibilities in action.

Listing 4.1 Some Valid Ways to Specify Superclasses

```
class A: pass

class B(A): pass # direct usage

import m
class C(m.A): pass # using an attribute

def foo(): return A
class D(foo()): pass # using the result of a function call

class E(B, A): pass # multiple inheritance

x = [B, A]
class F(*x): pass # multiple inheritance using list unpacking
```

Python also adds the concept of *metaclasses*. A metaclass is, to put it simply, a class that creates other classes. If regular classes can create instance objects, metaclasses can create class objects. If a class definition contains a `metaclass` keyword argument, after the definition is parsed and the corresponding class object created, that object is passed as an argument to the `__new__()` method of the declared metaclass, along with its assigned name, list of base classes, and some optional attributes. The output of that method, rather than the original object, is then bound to the specified name.

Metaclasses enable programmers to dynamically add additional methods and other features to a class while avoiding boilerplate, or they can replace the class definition with something entirely different. An example of declaring and using metaclasses is shown in Listing 4.2.

Listing 4.2 Metaclass Example

```
class A:
    def foo(self):
        print('A.foo')

class Meta:
    def __new__(cls, clsname, bases, attrs):
        return A

class B(metaclass=Meta):
    def foo(self):
```

```python
        print('B.foo')

# Because of the metaclass, B is now equal to A
b = B() # b refers to an instance of A
b.foo() # prints 'A.foo'
```

4.2 Function Invocation Rules

If we want to be able to create a better search for targets of function invocations, we must first understand the ways in which functions can be created and invoked. This section focuses on just that topic.

4.2.1 Namespaces and Scopes

Before progressing further, we need to introduce two concepts, *namespaces* and *scopes*. They will be important for the rest of this chapter.

A namespace is a mapping between names and objects. Whenever an identifier is used in a program, a namespace—or rather, a series of namespaces—is used to determine which object that identifier is currently referring to. Whereas in other languages namespaces can be thought of as a more conceptual term, Python gives them a literal meaning, implementing each namespace as its own internal dictionary. As we would expect, the mapping defined by each namespace is independent, and two namespaces can refer to two distinct objects by the same name.

Different syntactical elements create new namespaces at different times. A function creates its local namespace when it is called, and it ceases to exist at the end of the function's execution. The namespace of an imported module is created when that module is imported, and the one containing all built-in names exists as soon as the Python interpreter is started.

A scope is a textual region that uses a namespace. This means that that namespace is searched when an unqualified identifier is used within the corresponding scope. At any given time, up to four kinds of nested scopes are present:

1. The innermost scope, containing local names.
2. The scopes of any enclosing functions, which contain non-local, non-global identifiers.
3. The global scope of the current module.
4. The scope of built-in names.

When resolving a name, the namespaces associated with these scopes are searched in that order, with the enclosing function scopes searched from the innermost one outward.

This implementation has some consequences that may be surprising to someone unfamiliar with the language. For one, scopes and namespaces don't always correspond to syntactic blocks. In other words, assignment to a variable done within an `if` statement is reflected outside of it and may even overwrite a previous value associated with that name. Another potential oddity is that not all namespaces represent a scope. For example, names assigned within a class definition cannot be accessed without a qualifier from the body of a function declared within that class, as demonstrated in Listing 4.3

Listing 4.3 Scope vs. Namespace vs. Block

```
a = 0
class X:
    b = 0
    def foo():
        c = 0
        def bar():
            d = 0
            if True:
                e = 0
            print(a) # Works, accessible through the global scope
            print(b) # Fails, no scope uses the X class namespace
            print(c) # Works, nonlocal scope of foo()
            print(d) # Works, local scope of bar()
            print(e) # Works, also in local scope of bar()
```

The last notable consequence we will mention is that the names of built-in functions and classes aren't reserved; on the contrary, they have the lowest priority of all the scopes. Declaring a variable named `int` or `print` is perfectly legal and will effectively override the built-in identifiers with those names.

When assigning to a name, the local namespace is always used, even if that name was previously found in some parent scope. The only exception to this rule is when the `global` or `nonlocal` keywords are used with those names, in which case the assignment happens in the global or innermost nonlocal scope respectively, as seen in Listing 4.4

Listing 4.4 Assignments in Different Scopes

```
def foo():
    a = 0
    def local_assign():
        a = 1

    def nonlocal_assign():
```

```python
        nonlocal a
        a = 2

    def global_assign():
        global a
        a = 3

    local_assign()
    print(a) # 0
    nonlocal_assign()
    print(a) # 2
    global_assign()
    print(a) # 2

foo()
print(a) # 3
```

4.2.2 Function Declaration

A function is declared with a function definition statement. The definition consists of a header, which specifies the name of the function and its parameters, and a body, which includes the code to be executed when the function is run. When the interpreter encounters a function definition, it creates a new function object (as described in Section 4.1), which is itself a wrapper around the code object that represents the function body. This newly minted function object is then assigned to the local namespace with the name declared in the header. We can check that this is a regular namespace assignment for example by using the `global` keyword with a function:

```python
def outer():
    global inner
    def inner():
        print('inner')

outer()
# 'inner' is accessible in the global scope
inner()
```

Because function definitions are just statements and functions themselves are just regular named objects, they can appear anywhere a statement could appear. We can have functions declared inside other functions, in `if/else` blocks, inside class definitions, or directly at the top level of a module. Functions can be re-assigned with new definitions or even replaced with regular objects.

Aside from the name, the definition header also includes the list of parameters. A parameter can be one of five kinds, depending on how it can be assigned during invocation:

- *positional-only* Can be assigned only positionally, i.e. `foo(1)`.
- *keyword-only* Can be assigned only as a keyword argument, i.e `foo(arg=1)`.
- *positional-or-keyword* Can be assigned either positionally or as a keyword argument.
- *var-positional* Variadic parameter, specifies that any number of subsequent positional arguments can be passed.
- *var-keyword* Variadic parameter, specifies that any number of subsequent keyword arguments can be passed.

By default, all parameters are *positional-or-keyword*. To specify some subset of parameters as *positional-only*, a special "/" token is inserted after them in the parameter list. A *var-positional* parameter (commonly named `args`), of which there can be at most one, can only appear after all *positional-only* and *positional-or-keyword* parameters and is labeled by prepending a "*" before the parameter name. To specify some subset of parameters as *keyword-only*, a special "*" token can be inserted before them in the parameter list, or if a *var-positional* parameter is present, following parameters are *keyword-only* by default. A *var-keyword* parameter (commonly named `kwargs`), of which there can be at most one, can only appear after all other parameters and is labeled by prepending "**" before the parameter name. The combination of these rules can result in several forms of the signature, some of which are shown in Listing 4.5

Listing 4.5 Examples of Valid Parameter Lists

```python
def foo(pos_or_kw_1, pos_or_kw_2): ...
def foo(pos_or_kw, *args, **kwargs): ...
def foo(pos_only, /, pos_or_kw): ...
def foo(pos_or_kw, *, kw_only): ...
def foo(pos_only, /, pos_or_kw, *, kw_only, **kwargs): ...
def foo(pos_only, /, pos_or_kw, *args, kw_only, **kwargs): ...
```

Aside from the name, each parameter can declare a default value by appending "=" and an expression (e.g. `def foo(a=0): ...`). If a value for a parameter with a default value isn't supplied, the result of the expression is used instead. We should note that the number and kinds of parameters are irrevocably tied to the signature of a function, and aside from default or variadic parameters, a function call cannot supply more or fewer values than the parameter list requires. It is also notable that it is illegal for a *positional-only* or *positional-or-keyword* parameter to have a default value if it is followed by other parameters of those kinds that do not have one. This limitation does not apply to *keyword-only* parameters, which can be interleaved at will.

A function can also be decorated with one or more decorators (see Listing 4.6). After a function definition is processed into a function object, if any decorators are assigned to the function, the function object is passed as an argument to the decorator, and the result is bound to the specified name instead of the original object. In a sense, decorators serve a similar function to metaclasses described in Section 4.1.2, and in fact when applied to class definitions instead of function definitions, they can be in some circumstances used for the same purpose.

Listing 4.6 Decorator Syntax

```python
def positive_arg(func):
    def inner(x):
        if x < 0:
            return func(-x)
        else:
            return func(x)
    return inner

@positive_arg
def foo(x):
    return math.sqrt(x)

# Equivalent way of applying the transformation:
foo = positive_arg(foo)
```

4.2.3 Calling Functions

When a call expression is encountered by the Python interpreter, first the target of the call is found, using the scoping rules described in Section 4.2.1. Then, the supplied arguments are matched to the function parameters. To do so, first, the positional arguments are matched into the corresponding parameter slots. Then, any keyword arguments are put into their appropriate slots using their name. Lastly, if any parameters don't have an assigned argument, their default value is used, and any leftover arguments are collected into the variadic parameters. If any step of this matching fails, for example, because a non-default parameter wasn't supplied a value, an exception is raised.

One feature—shown in Listing 4.7—that further complicates this process is unpacking. When a star-prefixed collection is supplied in the argument list, it is unpacked and each of its values is taken as a separate positional argument. Moreover, if the collection is a mapping, it can be supplied with a "**" prefix, in which case each key–value pair it contains is taken as a separate keyword argument.

Listing 4.7 Argument Unpacking

```
collection = [2, 3]
mapping = {b: 2, c: 3}

def foo(a, b, c):
    print(a, b, c)

# These are all equivalent
foo(1, 2, 3)
foo(1, *collection)
foo(1, b=2, c=3)
foo(1, **mapping)
```

This makes matching arguments to parameters noticeably more difficult for any kind of static analysis. Especially when variadic parameters interact with unpacked arguments, there can be many different possible mappings, which we must all consider to ensure that no valid assignment is missed.

One last particularity concerns instance methods. When an instance `c` of class `C` is created, for each function defined on `C` (other than functions explicitly marked as static or class functions), an instance method object is created and bound to that instance. That instance method has the same name as its corresponding function and is accessible as an attribute of the instance. The one place where they differ is in their parameter lists.

Python has no concept of an implicit `this` value inside a method body. Instead, the first parameter is used to contain the instance upon which the method is being called (this parameter is conventionally named `self`). The instance method object does this by taking its parameter list, converting it into a new one with the first argument added to be the instance on which it is bound, and then calling its corresponding method. This way, `C.foo(c, x, y)` is equivalent to `c.foo(x, y)`.

Finally, we must mention that functions (and instance methods) are not the only callable objects in Python. We have already seen through examples that calling a class object creates an instance of that class. The instance is created by calling a static `__new__()` function (which is supplied any arguments passed to the class call) and typically initialized using the instance's `__init__()` method (which is supplied the newly created instance as `self`, as well as any arguments passed to the class call). Class instances can themselves be callable if they define a special `__call__()` method, though this is seldom used.

4.3 Type Inference Mechanisms

Despite its challenges, some techniques for generating type inference in Python code have been developed. Since this work treads in similar waters, we would like to use this section to look at some of these techniques and consider their usefulness with regard to our problem domain.

4.3.1 Hindley–Milner Type System

Well-known and extensively studied, Hindley–Milner [7] is a system of typed lambda calculus that has the ability to determine the most general type of any expression in a given program. Built-in type inference features based on this system are available in several programming languages, most notably in functional languages such as ML or Elm. These languages are a natural fit for this approach, as their structure more closely resembles the typed lambda calculus Hindley–Milner is based on.

The quick gist of how type inference works in Hindley–Milner is by creating a set of rules that specify some consequence of an expression in a given context, then applying those rules until every expression in the program has been labeled. We shan't go into much more detail, because as we found, this isn't particularly useful in our case.

The type system of Python is vastly different from that of a statically typed functional language. We would have to deal with issues such as heterogeneous collections, dynamically changing types of variables, and OOP features such as subclassing, none of which this type system is inherently equipped to handle. Moreover, its high theoretical complexity would make any implementation—much less a highly customized one required for Python—extremely difficult to build and maintain. This combination of factors makes it a subpar choice for our use case.

4.3.2 Aggressive Type Inference

As a relatively obscure option, *aggressive type inference* [8] is a proposed method initially conceived as part of a translation tool from Python to Perl. The main idea behind this approach is that although Python is nominatively a dynamic language, most programs don't actually make use of this dynamicism all that often. We can therefore make a fairly well-informed estimate of the types of a variable based on just a few simple rules:

1. Flow insensitivity. Itself a core tenant of data flow analysis, it basically means that the algorithm ignores any control flow structures and considers every possible path through a procedure. A fitting rule for usage in an application focused on analyzing data flows.

2. Type consistency within a scope. This rule codifies the core thought outlined above. When a variable has some type T during its lifetime, it is considered to have that type at every point in the scope where it is bound to a value.

To look at a practical example of how these rules work together, let's consider the following method:

```python
1 def foo(x):
2     print(x)
3     if input():
4         x = 123
5     else:
6         x = "foo"
7     print(x)
```

By the rule of flow insensitivity, the algorithm would determine the type of `x` to be the union of `str` and `int` on lines 4, 6, and 7. By the rule of scope consistency, it would also apply the same type to `x` on line 2. The implementation of this approach could be fairly quick and straightforward and potentially produce consistent results in a large percentage of situations. However, there are also some problems we have to consider.

This method does not appear to have been expanded upon in a significant capacity since its introduction, and the literature covering it is very scarce. This includes any sample implementations—while the author reportedly developed an implementation capable of generating type inference this way, that implementation doesn't seem to have been made available, meaning we would have to develop our own version using only the description provided in the paper. Lastly, the rule of scope consistency is an issue. While most code may adhere to the "mostly static" limitations as outlined, we are operating in an area where over-approximation is always preferable to the loss of information, which is what we risk when making that assumption. There will be situations where a variable will completely change types within a scope, and we must be able to detect and handle those situations. In other words, we cannot afford to be as aggressive as this method suggests. Since this is a load-bearing part of the whole process, it renders aggressive type inference impractical for the scanner.

4.3.3 Pyan

Pyan [9] is a library that aims to statically generate a call graph of a Python program that boasts some impressive features. Unfortunately, while the appeal of using a ready-made solution is clear, there are some glaring problems with using it in our project.

The last supported version of Python is 3.6, and based on the communication from the developer, there don't seem to be any current plans to develop the project further. Furthermore, it is itself written in Python. Since Manta Flow is distributed as a Java binary and may run on computers without a Python runtime, this poses a distribution challenge. Finally, it is licensed under the GPL-2.0 license, which would make it incompatible for use in a closed-source application.

4.3.4 MaxSMT-Based Type Inference

`Typpete` [10] was created as a novel approach to generating type annotations for Python 3 programs. This program encodes the type constraints as a MaxSMT problem, then uses an off-the-shelf SMT solver such as Z3 [11] to generate solutions for that problem. According to the authors of this tool, the results of this tool are competitive and scale to real-world programs.

The issues we encounter with this solution are similar to those we have seen before. It makes some restrictions on what language constructs are acceptable and disallows e.g. heterogeneous collections and dynamically changing the type of a variable throughout the program. It would also require adding a whole SMT solver into Manta Flow, which seems somewhat excessive for just this one use case, it doesn't support newer versions of the language, and the licensing is ambiguous.

4.3.5 Machine Learning

There have been some recent advancements in type inference methods based on deep learning methodologies. For example, Saifullah et al. [12] propose a technique that, based on their experimentation, achieves up to 80 % precision in correctly determining types in analyzed code with over 70 % recall.

While we would be remiss not to mention this group of tools, for the uses we envision in our work with Manta Flow, we must reject machine learning at the outset. Partially because of the lack of familiarity the team as a whole has with such techniques and the lack of visibility into the results that we would expect as a consequence. For the most part, though, it is because these methods are necessarily probabilistic, and even though the confidence of any particular result may be high, we simply cannot afford to misrepresent the type of an expression and possibly mischaracterize the data flows connected to it as a result.

Chapter 5

Overall Design

In the previous chapter, we explored some type inference mechanisms and whether they fit our needs. To understand one reason why they didn't, we must realize that the goals and constraints of general-purpose type inference algorithms tend to slightly differ from ours. The question they aim to answer is, roughly, "Given a piece of possibly incomplete code, what type information can we assume about it?" (In this context, we can imagine "possibly incomplete code" as, for example, a function definition without any invocations or a module that accesses a locally undefined global variable).

In our analyzer, we have complete information about the code we analyze. Every argument of every function invocation has been previously defined somewhere, we have access to the full code of every imported module[1]. This simplifies our task. Our objective is also different; rather than inferring types for general use or for checking correctness, we are interested purely in increasing the precision of the analysis of data flows inside the program. To put it another way, we don't care about the most general type a function could possibly return; we only care about which types it returns in the invocations we specifically analyze.

5.1 Note on Precision

While we obviously want to reduce the number of possible targets of function calls as much as we can, it is important to realize that we do not have to aim for perfection. Since we're performing type analysis largely as an optimization measure, there can be approaches that would produce more specific results in certain situations, but the performance and complexity of those approaches would disqualify them from being useful. This fact has direct consequences on our design.

[1]With the exception of plugin-handled libraries, which require special treatment.

There may also be situations where we are unable to correctly infer type information for an expression, or where an expression could literally be of any possible type. Because we always prefer over-approximation over loss of information, in cases where the target of a function invocation depends on such an expression, we can (and must) fall back on the existing approach of matching function names and parameter counts. To identify such cases, our type system must include an explicit type denoting that no information is available. This type is henceforth referred to as `Any`.

There are also some limitations that arise from any static analysis of Python, and we should note situations that we can definitely not take into account upfront. Unsurprisingly, these will be situations where some form of dynamic code generation is performed. Functions such as `eval`, `exec`, or `__import__` are not generally statically analyzable, and while partial support could be added e.g. in situations where they are called on a string literal, those don't tend to be the situations where these functions are used in the first place. Also unsupported are constructs such as declaring classes through a `type()` call or changing inheritance by setting the `__bases__` property. The ways to dynamically mess with the structure of Python classes and functions are too numerous to cover fully, suffice it to say that if a program is doing something really strange, it's probably not supported.

Support is also currently not assumed for decorators and metaclasses, as those are as a whole not supported by the scanner at the time of writing.

5.2 Main Idea

We are using a process we shall call *object origin tracking*. The thought behind this process is simple: we have all the application code available to us. For every function call, we know the arguments passed into it. For each assignment, we know what expressions are on both sides. Using this information, we can track known types of any variable—in fact, of any expression—throughout the program, much in the same way that we track data flows.

Let us recall the worklist algorithm introduced in Chapter 2. The scanner processes one executable, propagating the data flows throughout its execution, then adds all the callers and callees of that executable to the worklist. The same approach can work for propagating known types. Starting with the instruction during which an object is created (hence the name), we track the object throughout assignments, function calls, and other operations to figure out what types are contained in the result of those operations.

One benefit we gain with an approach that closely resembles the inner workings of symbolic analysis is the possibility of the reuse of a lot of the existing infrastructure in the scanner. Instead of creating the solution wholly from scratch, for some parts we only need to generalize existing components, which makes the implementation simpler and more maintainable.

The similarities between these two concepts even led to a previous effort that tracked so-called *type flows* directly alongside data flows during symbolic analysis. However, the inflexibility of this solution—as well as the fact that symbolic analysis was already by far the most complex part of the scanner—led to the inevitable outcome in which these type flows were never successfully used and had been removed from the scanner before this work began. Instead, we propose type analysis as a separate preliminary step that happens before symbolic analysis but can inform it. Chapters 6 and 7 provide a detailed explanation of how type analysis was implemented.

To use any gathered information for a better search of possible targets of function invocations, we construct a revised call graph. The call graph will be based on the types gathered during type analysis and, for instance functions, will only find invocations on types found in those instances. By converting the type data into a call graph, we utilize an interface through which symbolic analysis can yield the benefits of type analysis without having to know anything about it—the new call graph is simply injected as a dependency, just as the old one was. Chapter 8 documents the implementation of this new call graph.

5.3 Possibilities of Type Propagation

In order to successfully track expression types throughout the program, we must examine how they can change during the course of program execution. There are only two situations which can affect the type of an expression.[2]

- When a value is assigned to an expression, the type of the assigned value is added to the list of possible types for that expression.
- At the end of a control flow segment (e.g. an `if-else` block), the possible type of an expression becomes the union of all types determined in each branch of that segment. (A union of any type and `Any` can be simplified to `Any`.)

This section describes how different types of assignments can be processed.

[2]Since we're using the worklist algorithm, we can never remove or replace possible types from an expression, as that could cause the worklist to run indefinitely. We can only add new ones.

5.3.1 Literal Value Assignment

The most straightforward case is when a literal value is assigned directly to an expression. In those cases, we can clearly determine that the expression can have the type of that literal. For example, `x = 2` or `x = ["a", "b", "c"]` let us infer the possible type of `x` as `int` and `list[string]`, respectively.

5.3.2 Constructor Call Assignment

When a new object is initialized we can determine the type of that initialization based on the invoked class. For example, `x = A()` means `A` is now a possible type for `x`.

5.3.3 Expression Assignment

When one expression is assigned to another (as in `x = y`), the type of `y` is added to the list of possible types for `x`.

5.3.4 Function Call

The interesting case occurs when we assign the return of a function invocation into an expression (`x = foo()` or `x = y.foo()`). To determine the return value of such an invocation, we must first resolve the target of the invocation itself, and only once we do that can we actually process the resolved function.

Moreover, the return type can vary between invocations. If a function or method accepts arguments, it can change its execution almost arbitrarily based on the values passed into those arguments. The simplest example of this is the identity function, which merely returns its input as an output—the return type of such a function is wholly dependent on the type of its argument. Aside from argument values, functions can also vary their flow based on the current state of the program (for example by checking a global or instance variable), the result of another function call, etc.

Given these complexities, it seems that in the general case, we have no better option than to take the information we have about each invocation and the provided arguments and use that to determine the correct return type.

It is here that the worklist algorithm comes in handy. When encountering a function call, we can use whatever data was gathered for that function in this context. We then schedule the function into the worklist, and any change in the flows produced by the function will trigger another iteration of processing its callers until we reach a fixed point. This way we can be sure that each expression eventually gets all the data that was possible to find for it.

It is worth noting that this is the one place where we could theoretically be aided by an existing general-purpose type inference algorithm; producing a general type of a function could alleviate some of the burden of this shifting nature of function invocations. However, after careful analysis of the existing approaches, it was deemed that their inclusion would be too complex and costly for the potential benefit they might bring.

5.3.5 Plugin-Handled Functions

So far, we have been operating under the assumption that all code utilized by the analyzed program is directly readable during the analysis. There is, however, one situation where this assumption does not hold, and that is when dealing with functions handled by plugins. For specific libraries where full implementation either is not available (due to being written in another language), or its inner complexities significantly overshadow its public API, we emulate calls to those libraries through the use of predefined propagation modes. In order for type tracking to function properly for these functions, their propagation modes also have to add runtime type information to their return value.

In the simplest case, this information could be declared purely as an enumeration of possible literal types. To accommodate basic functionality, we should also extend such an enumeration with at least references to the type of one of the arguments. For example, being able to define that a method can return `self` is a prerequisite for any API that supports method chaining.

If it was deemed necessary, this declaration could be extended with some more general kind of reference (i.e. a method could be declared as returning the same type as another method with the same arguments). In the most generalized version, we could even implement some sort of decision tree that would accurately describe the return type based on the types (and possibly values) of the arguments passed, though such generality seems unnecessarily ambitious.

To avoid any potential loss of information, any function that does not have a defined return type must be implicitly assumed to return `Any`.

Details of the implementation that solves these issues are present in Chapter 7.

Chapter 6

Tracking Runtime Types

This chapter describes how we implemented the tracking of types in the scanner. The first section provides a more detailed explanation of what happens during symbolic analysis, and the rest of the chapter shows how we utilized and extended that process for the purposes of type analysis.

6.1 Symbolic Analysis Architecture

As previously alluded to, the core of the symbolic analysis is a worklist algorithm. When describing the worklist algorithm in Section 2.2, we spoke of the way new entries are added to the worklist based on invocation contexts, but little attention was paid to the way the executables are actually processed. To understand how we can utilize the existing infrastructure for type analysis, we first have to understand how it works in symbolic analysis.

6.1.1 Execution Simulator

We do not want a static analyzer to work directly with code (or directly with the parsed statements created from the code). Such an analyzer would have to be concerned with unnecessary detail about the exact representation of the program, and it would obfuscate the analysis logic behind minutiae around the specifics of handling expressions.

Instead, an intermediary layer is introduced that creates an abstraction over this representation and, in essence, turns the parsed tree of statements into a linear stream of events to be used by the analyzer. This layer is called the execution simulator (or just *simulator* for short).

The simulator operates on symbolic expressions, meaning it does not care about any values of the expressions it processes. Each kind of expression or state-

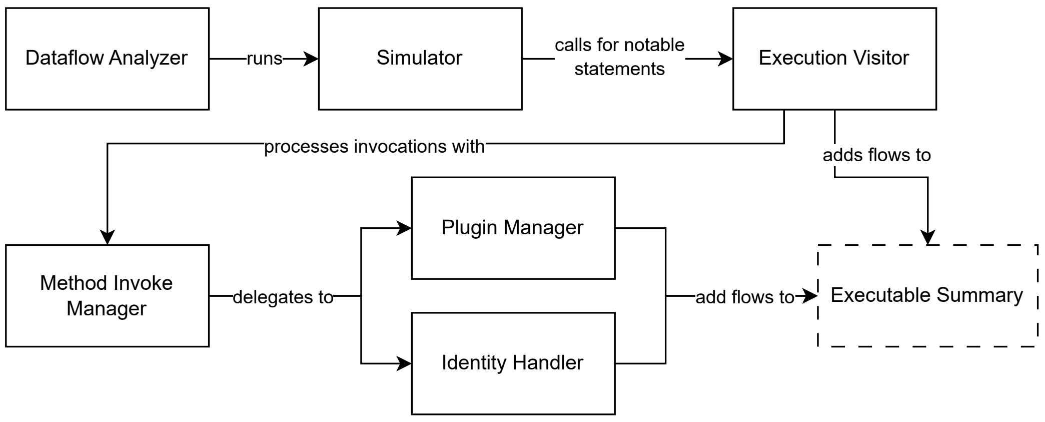


Figure 6.1 Simplified Diagram of Processing One Executable During Symbolic Analysis

ment the simulator handles has a separate handler registered, which contains the logic for simulating that kind of statement. For example, handlers for compound statements (statements that contain a block of statements, such as a `for` loop) are programmed to simulate the outer expression (the loop condition), and then in turn each statement present within the block.

There are also special expression-transforming handlers, which convert the parsed expressions created during the input processing stage (per Section 2.2.2) into a different representation that is more suitable for the analyzer. This is another way that the simulator makes implementing the analysis simpler.

6.1.2 Execution Visitor

Not all statements and expressions processed by the simulator are relevant to the analysis. We know that data flow analysis is context-independent, for a start, which means we do not have to care that much about the starts and ends of control flow segments. Conversely, when a relevant statement does appear in the simulator, we need some way of notifying the analyzer about it. This job is performed by the *Execution Visitor* interface. Whenever the scanner needs to process something using the simulator, it injects a visitor instance as a dependency, and this dependency is then called whenever suitable expressions are found.

There are three execution visitor implementations in the scanner. Alias analysis uses two to collect all assign statements and return statements respectively. Symbolic analysis uses its own visitor to implement the logic of how data flows are created and propagated during the processing of a single executable. Note that the visitor interface has default empty implementations for all statements, so implementations can provide overrides only for the methods that they are interested in and ignore the rest.

6.1.3 Flows and Flow Sets

There are many types of flows in the scanner, depending on what data needs to be propagated. When `x = "str"` is processed, a `ConstantFlow` containing the literal is associated with `x`. A dictionary may have several `DictionaryItemFlows` describing its contents. Even function and class definitions produce a type of flow.[1] The set of flows associated with any given expression is called a flow set.

The most important flows, however, are the kinds that represent some movement of data. Reading from a file will result in a `FileReadFlow` being created, which will contain a `FilePathFlow` describing which file was accessed. Similarly, writing to a file produces a `FileWriteFlow`.

While these are the flows we are ultimately interested in, we have to track the others as well for the scanner to be at all useful. Consider the following simple example:

```python
x = "file.txt"
with open(x) as f:
    y = f.read()
```

Only by tracking the string literal that is assigned to `x` can we determine which file was opened on the following line.

For every invocation context of every executable we analyze, a mapping between expressions in that executable (including its parameters) and their flow sets is maintained. This mapping is called the executable summary.

To be able to actually resolve the path of some data flow from beginning to end, each "data sink" operation (i.e. a write of some kind) is also tracked separately in the executable summary. The "important" flows also track their origins. For example, if we read from a file to a variable `x`, associating a `FileReadFlow` with it, and then we write that variable to a different file, the `FileWriteFlow` that results will include a pointer to the read flow that was included in the flow set of the written expression. When generating the final linage graph, we can look at all the end operations we collected, then walk back through these links to obtain the full path.

6.1.4 Method Invoke Manager

Invocations hold a unique place in the context of the analysis, as they represent the connections between different executables. They must be handled uniquely as a result. When processing an invocation expression, the analyzer finds all possible targets for the invocation, then for each performs the following:

[1]Though those flows are not currently used for anything.

- The invocation is recorded (for worklist purposes).
- The executable summary of the target is found.
- The flows currently associated with the parameters of the function are propagated to the arguments of the invocation.
- The flows currently associated with a `return` expression of the function are propagated to the invocation itself.
- Any `nonlocal` and `global` scoped variables are propagated to their outer scope if necessary.

At least, that is what happens for application functions. Basing flows on a `return` expression only works when there is a `return` expression that we can see, i.e. when we have the code of the function available. For functions which are defined via plugins, or even totally unknown to the scanner, we need a different strategy.

The `MethodInvokeManager` class is responsible for ensuring all invocations are handled correctly based on the semantics of the target. For application functions, it completes the steps outlined above. For plugin-handled functions, it delegates them to the plugin manager, which is able to apply the correct propagations based on the provided configuration for the function.[2] And for functions that do not fall into either of those categories, a specially designated *identity handler* is used.

The identity handler is a consequence of the idea that the scanner should try not to lose potential data flows, even when it does not have all the relevant information. When an invocation of an unknown function is found, be it because it is from a library the scanner does not have a plugin for or because it was defined using a construct the scanner does not support, we have no notion of what the effect of that function might be. The best we can do, then, if we do not want to lose all the potential flows, is to simply take flows from all arguments of the invocation and pass them into the return value, which is precisely what the identity handler does. For single-argument invocations, this is equivalent to applying the identity function, hence the name.

While better than simply dropping flows, applying the identity handler can still spoil the result of the analysis, especially when combined with an imprecise search for invocation targets. When the possible targets of an invocation are found to be both a function that we can process regularly as well as one that has to be processed using the identity handler, we have to apply both, effectively

[2]More on this process in Chapter 7.

mixing the unprocessed arguments into the result. This can then have a cascading effect for the rest of the analysis.

To determine how the `MethodInvokeManager` should process any given call, each function is assigned a *processing mode* during infrastructure analysis, which then informs how to handle said function. There are four main processing modes:[3]

- *Application-Handled*—The function is a regular application function for which we can analyze the source code directly.
- *Plugin-Handled*—The function has defined propagations in a plugin.
- *Skipped*—The function is configured in a plugin to explicitly note that its result is not relevant to data flow analysis and no propagations should be applied for it.
- *Identity-Handled*—The function should be processed using the identity handler.

6.1.5 Data Flow Analyzer

Now that all the component pieces have been introduced, we can show how they are combined. The `DataflowAnalyzer` class is the top-level class through which symbolic analysis is run. After some initial setup, it begins the worklist algorithm, starting from the entry point of the analysis (see Figure 6.2).

In each iteration, it takes one invocation context from the work list, copies its execution summary, then runs the simulator on the invoked executable.[4] After the simulation is finished, it stores the resulting execution summary in a map under its invocation context.

If the summary changed in any way (meaning new flows were added), the worklist is updated. We add the invocation context of each registered caller and each registered callee, and we also re-add the context we just processed. Doing so ensures that the analysis stops only once a fixed point is reached. If any new invocation contexts were created during the simulation, they are always added to the worklist, even if the flow of the current function did not change.

The result of the analyzer is the collection of invocation contexts and their corresponding executable summaries that were encountered during the analysis.

[3]There is one more processing mode, *group-handled*. Its semantics are a bit more complicated and we explore them in detail in Chapter 9.

[4]Remember that only application executables enter the worklist, calls to plugin functions are propagated directly.

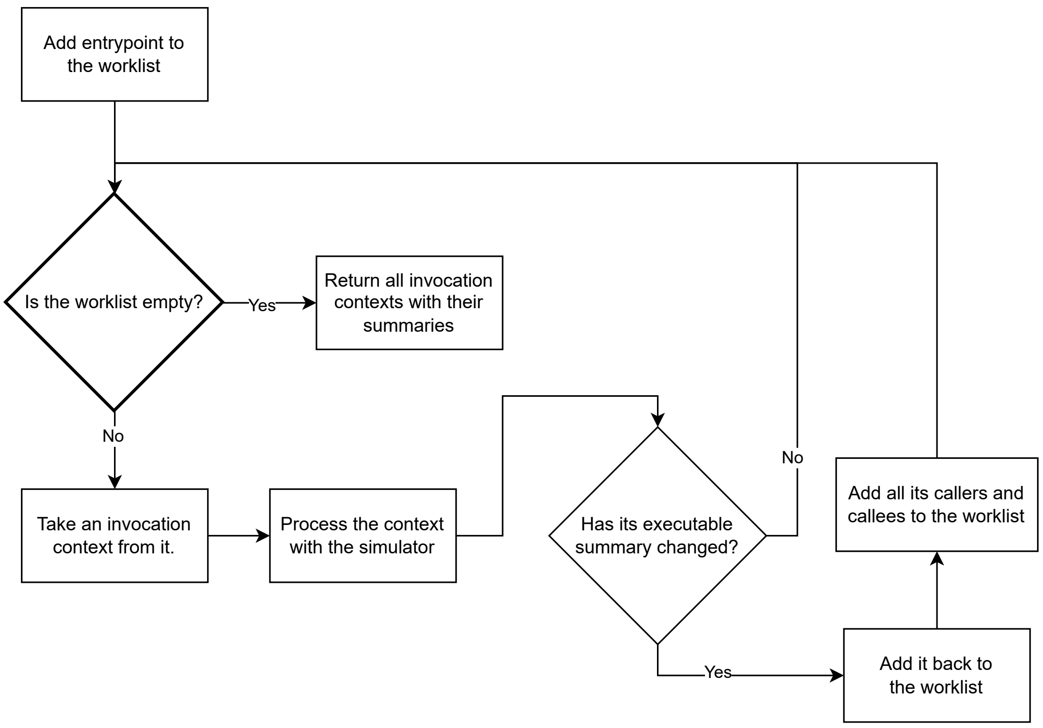

Add entrypoint to the worklist
Is the worklist empty?
Yes
Return all invocation contexts with their summaries
No
Take an invocation context from it.
Process the context with the simulator
Has its executable summary changed?
No
Yes
Add all its callers and callees to the worklist
Add it back to the worklist

Figure 6.2 Worklist Algorithm Flowchart

6.2 Reusing Existing Infrastructure

In Chapter 5, we introduced the idea that due to the way we approach type analysis, we want to utilize as much of existing code from symbolic analysis as possible. We initially considered only the reuse of certain components, such as the worklist and the concept of flows. However, after further consideration, we discovered that the overall structure of type analysis would very closely mimic that of symbolic analysis.

Instead of implementing these structures separately, then, which would result in large amounts of code duplication, we therefore directly utilize `DataflowAnalyzer` for type analysis, parameterized in a way that enables us to use it for our purpose. That way, the crux of type analysis will run using exactly the same codebase as symbolic analysis, with changes made only where necessary (and of course with different semantics). This realization somewhat shifts the focus of this work. Rather than describing only the additions we've made to implement type analysis, significant attention must also be given to what was changed in the existing codebase.

Much of the code used by symbolic analysis is what we might call "glue". We use this term to describe code that is necessary for the analysis to work but is not specific to the particular semantics of symbolic analysis. In this group, we include for example all classes related to administering the worklist, calling the simulator on invocations, creating and managing invocation contexts, handling variable scopes, and so on. All such code is available for reuse to type analysis.

We identified three parts as being specific to symbolic analysis:

- The flows. While the general concept of propagating flows can be reused for type analysis, the actual implementations of flows that are created in symbolic analysis are not reusable.
- The execution visitor of symbolic analysis, `ExecutableAnalysisVisitor`. This class implements the logic of how different statements are processed during symbolic analysis. As such, it cannot be used for other purposes.
- The plugin handlers. Plugin-handled functions use specific propagation modes to create flows. The results of these propagations are specific for symbolic analysis—because they produce the flows used by symbolic analysis if nothing else—and a separate way to manage those functions is necessary for type analysis.

These findings give us a natural way to parameterize `DataflowAnalyzer`:

- New classes for type flows are created.
- A new execution visitor specific to type analysis is implemented, which propagates type flows as described in Section 5.3. A new dependency to `DataflowAnalyzer` is added that lets us specify which visitor instances are used.
- A new set of plugins for type analysis is created. `DataflowAnalyzer` is already parameterized by the plugin manager, so type analysis needs only to create a new one containing this new set of plugins.

After these adjustments, we are able to generate executable summaries for either type analysis or symbolic analysis by running `DataflowAnalyzer` with the appropriate parameters set. The rest of this chapter describes in detail how each of the first two changes are implemented. The implementation of new plugins is then discussed in Chapter 7.

6.3 Type Flows

The flows used in symbolic analysis are all implementers of the `PythonFlow` interface. This interface defines only a few basic methods, all of which have default

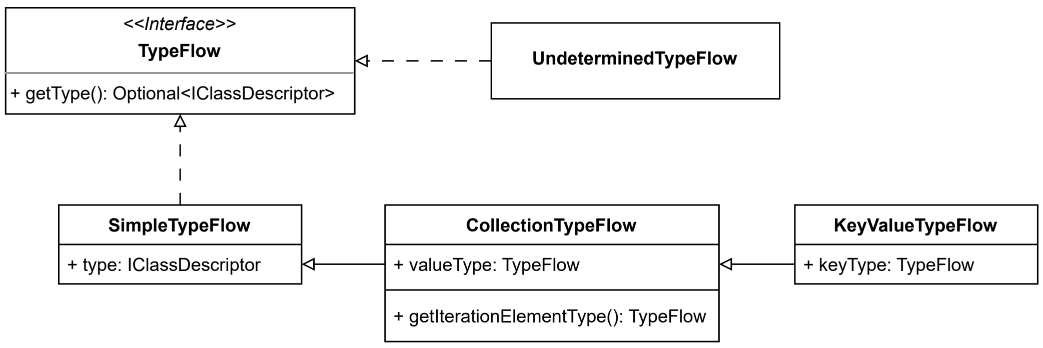


Figure 6.3 Class Diagram of Type Flows

implementations and are therefore usable in type analysis. We implemented `PythonFlow` with our own abstract base class, `TypeFlow`, from which all other flows inherit (see Figure 6.3). This allows all structures that expect to work with flows—such as flow sets or executable summaries—to seamlessly work with type flows as well.

Every type flow needs to store the information about which type it represents. Since each executable—and in particular, each class—is assigned a descriptor in infrastructure analysis, we use these class descriptors to determine the type stored in the type flow.

For the most basic type flow, aptly named `SimpleTypeFlow`, this is all it stores. Not all types can be represented that simply, though, and one outlier in particular would cause problems if we tried to do so—collections.

6.3.1 Collection Types

Let us imagine the following snippet:

```
l = [1, 2, 3]
x = l[0]
```

We see that `l` is of type `list`, as it was just assigned a list literal. What is the type of `x`? If those two lines follow each other, then we can tell it is `int`, but there is no way to make that determination just based on the fact that `l` is a `list`. We need a way to also store the inner types stored within the collection. In the parlance of Python, we could more accurately describe the type of `l` to be `list[int]`.

To that end, we created a second class of type flows, `CollectionTypeFlow`. This class is a subclass of `SimpleTypeFlow` because every collection type is also a regular type. If we called `type(l)`, the answer we would get is just `list`. In addition, it stores a `TypeFlow` instance that describes which inner type it contains.

We need this to be represented by a type flow rather than just a second descriptor since we have to support nested collections as well—`list[list[list[int]]]` is a perfectly valid type signature.

Python collections are also heterogeneous, and the inner type of one can change at any moment, as shown by Listing 6.1. To simplify things, we do not try to store all of the inner types within one collection flow. Instead, for each inner type, we create a wholly separate type flow that has that inner type. To put it another way, we take advantage of the fact that for our purposes, `list[int, str]` is equivalent to `list[int], list[str]`.

Listing 6.1 Collections Changing Inner Types

```python
lst = [1, 2, 3]      # list[int]
lst.append("foo")    # list[int, str]
lst.pop()            # list[int]
lst += [2.0, "foo"]  # list[int, float, str]
lst.clear()          # list
```

To see why this assertion is true, let us examine the two main operations we want to perform on a flow set containing type flows. Firstly, we might want a set of all possible types of the expression that is associated with that flow set. This is the same as getting the class descriptor that is associated with each flow in the set, and in the aforementioned example of `list[int, str]` versus `list[int], list[str]`, both interpretations give us the same answer, `list`; twice in one case, but that is not a problem. Secondly, in case the associated expression is a collection, we might want a set of all possible inner types contained within that collection. This is the same as getting the inner type flow from each flow in the set, and then getting its class descriptor. Again, both cases provide the same two answers: `int` and `str`.

This representation can lead to counter-intuitive results when used with nested collections. The flows of such a collection, say `[(1, 2.0), ("foo", 3)]`, would result in the following set of flows:

```
list[tuple[int]], list[tuple[float]], list[tuple[string]]
```

This happens even though none of the individual flows exist by themselves as types within the expression, and we even lose the connections of which innermost types are together in the same tuple. Nevertheless, applying the same two main operations, even recursively, we still always get the correct result. This is in large part because we process expressions symbolically, so the actual positions of elements within a collection are irrelevant; whenever an expression returns one element, we must consider all of the possible inner types for it, whether it is accessed using a complex sub-expression or a constant index.

6.3.2 Mapping Types

There is a special kind of collection in Python called a key–value collection (sometimes also called a mapping). Not only do such collections contain values, they contain explicit key–value pairs. Accessing such a collection using a key gets you its corresponding value. The `dict` class is the prototypical example of a mapping inside the standard library.

For mappings, we need to remember both the key and the value associated with the flow. Thus, a new `KeyValueTypeFlow` class is introduced. It is a descendant of `CollectionTypeFlow`, as every key–value collection is conceptually a collection. In addition, it adds a `key` field that contains the type flow associated with a particular key–value pair.

There is an argument to be made that all collections are actually key–value collections. What is a list, if not a particular kind of mapping where all the keys are integers? This is in fact the view taken by symbolic analysis, where only one kind of flow exists to represent lists and dictionaries alike. The reason type analysis explicitly separates the two is twofold.

First, it conceptually tracks more closely the way these types are represented in Python. A list is not defined as a variant of a dictionary, it is its own type. Even the `typing` module, which provides definitions for type hints used in the standard library, contains separate `Collection` and `Mapping` definitions. The second, much more pragmatic reason, has to do with iteration.

When a collection such as a `list` or `tuple` is iterated over, it returns all of its values in sequence, as expected. This is illustrated in Listing 6.2. When a mapping such as a `dict` is iterated over, it returns all of its *keys* instead. To help differentiate the two, a `getIterationElementType()` is defined on `CollectionTypeFlow` and overridden on `KeyValueTypeFlow` to return the inner value and the inner key respectively.

Listing 6.2 Iteration of Collections and Mappings

```python
l = [1, 2, 3]
for element in l:
    print(element)    # prints 1, 2, 3

d = {"a": 1, "b": 2, "c": 3}
for element in d:
    print(element)    # prints "a", "b", "c"
    print(d[element]) # prints 1, 2, 3
```

6.3.3 Unknown Types

There are situations that type analysis explicitly cannot handle. Some library function may not have any configured propagations (recall the identity handler from Section 6.1.4). Some functions may be declared using syntax we do not support, or may be accessed through a superclass definition we are unable to parse. For such situations, we need to mark the flow set accordingly and fall back to the older, less precise way of finding invocation targets where the associated expression is involved.

The `UndeterminedTypeFlow` class serves this purpose. It is a singleton and does not contain any data. Instead, it serves as a marker that we are unable to properly figure out the possible types of an expression. In a sense, it is a semantic equivalent to Python's `Any` type, representing that something could be literally anything.

Once `UndeterminedTypeFlow` is in a flow set, it is pointless to add any further flows to it. A union of `Any` and anything else is just `Any`. It can still be used as a component of other flows without issue, though, with signatures such as `list[Any]` and `dict[str, Any]` being both possible and meaningful.

The existence of this type flow means that the getter for returning the descriptor associated with a flow returns an `Optional` instance rather than the descriptor itself. For all other flows, it can be unwrapped to get the underlying value. For `UndeterminedTypeFlow`, an `Optional.EMPTY` value is returned, indicating no class descriptor is associated with this flow. Coincidentally, this is the way we can detect whether this flow is present in the flow set.

6.4 Type Analysis Execution Visitor

The execution visitor used for type analysis is implemented in a class named `TypeInferenceVisitor`. This class is responsible for creating and propagating types correctly throughout the flow of a function. Here are the statements it handles.

6.4.1 Tuple Definition

```
(1, 2, "foo", 4, (a, "b"))
```

This expression is handled straightforwardly. From its parsed representation, we get a list of its items. For each item in this list, we create new `CollectionTypeFlows` of the `tuple` built-in class that contain the flows in that item's flow set. The newly created collection flows are then added to the flow set of the tuple definition.

We should note that list definitions are not handled explicitly. Instead, the parsing service decomposes them by transforming the initialization from `[1, 2, 3]` into the equivalent `#__builtins__.list((1, 2, 3))`, turning them into a tuple definition and a call to the `list` constructor.

6.4.2 Dictionary Definition

`{"x": a, "y": b, "z": 4}`

Dictionary definitions are similar to tuple definitions. We get a list of key–value pairs from the parsed representation of the code and create a new `KeyValueTypeFlow` of the `dict` class for flows associated with those pairs.

6.4.3 Assign Statement

`x = a.b`

We take all flows associated with the expression on the right-hand side and propagate it into the flow set of the expression on the left-hand side. We do the same for all aliases of the left-hand side that were found by alias analysis.

6.4.4 Return Statement

`return x`

We create an instance of a specially designated `ReturnExpression` class. We then propagate the flow set of the expression being returned (x) into its flow set. This class is used to find flows that are propagated as the result of an invocation of this function.

6.4.5 Invoke Expression

`x.foo()`

To process an invocation, we first need to find its possible target. Here we seemingly run into a kind of chicken-and-egg problem, since one of the goals of type analysis is to create a better way of resolving invocation targets. However, our usage of the worklist algorithm allows us to solve it in a way that we cover in Chapter 8.

For each target found, we create a mapping (or rather, mappings—remember the unpacking feature from Listing 4.7) between the arguments of the invocation and the parameters of the executable. This mapping, along with the flow sets of the arguments and information about the caller, is used to create a new invocation context for this call. That context is then passed to the method invoke manager, which facilitates the actual propagation of flows resulting from that call.

Special handling is required for targets that are classes, i.e. for invocations that create a new class instance. In those cases, we check if the class has a constructor declared on it, and if so, we handle it as though it was the constructor that was being invoked. Constructors do not have any return statements, but that is not an issue—we already know that the result of such an invocation will be a new instance of the class being created, so we can add a type flow of that class directly to the flow set for the invocation.[5]

6.4.6 Other Statements

The five kinds of statements described above are all we need to analyze the entire application. This is mostly thanks to two reasons. First, the context-insensitivity of the analysis means we do not need to care about control flow statements. Second is the fact that most of the expressions available in Python are transformed into invocations during parsing. We do not need special logic for `a + b`, because it is converted into `a.__add__(b)`. This transformation is possible for just about every operator in the language, saving us work we would have to do by implementing them all explicitly.

6.4.7 Inline Flow Generator

We have not yet considered how we process expressions such as numeric or string constants. Consider, for example, the tuple definition `(1, 2, "foo")`. The first step in processing it is to take the flows associated with each item in the tuple. We would imagine that this would result in simple type flows of `int` and `str`, but where do those flows come from? There is no visitor method for processing constants, and we cannot replace a constant with a method call (not without using another constant, anyway).

It is important to understand that such expressions are useless to store in the executable summary because their flows can always be computed ad-hoc as needed. No matter where we write `"foo"`, its value (and its type) will always be the same. For this reason, the scanner introduced the concept of *inlined expressions*.

When propagating flows from some expression, the scanner does not just look up the flow set and do a simple copy. It also checks the type of the source expression, and if it is one of these inlined expressions, it creates the inline flows for them based on their value.

This used to be hard-wired into the propagation code and created flows used in symbolic analysis. For example, `1` would correspond to a `ConstantFlow` with

[5]The constructor still has to be processed though, because it may create e.g. some properties on the created instance.

a value of one. That does not work for type analysis, where the desired flow would instead be a `SimpleTypeFlow` of type `int`.

To address this, we extracted that logic into a separate abstract class named `InlineFlowGenerator`. This class is responsible for generating flows on the fly for string and numeric constants, as well as the unary `not` and the individual key–value pairs in a dictionary definition. Each execution visitor provides its own implementation of this class when propagating types, which deals with the specifics of what flows should be included for inlined expressions.

As we said, we do not want to have these flows waste space in the executable summary. That is why, at the end of every method in the execution visitor, a clean-up operation is performed that prunes all inlined expressions from it.

Chapter 7

Plugin-Handled Functions

The newly added execution visitor can work with application code that is run through the simulator, but it has no ability to handle functions that are only configured through library plugins. This chapter describes how we added type information to such plugins, as well as an alternate approach that was considered, but ultimately rejected.

7.1 Type Hints

This section describes a considered approach that, while ultimately rejected, is nevertheless worth discussing.

Since Python 3.0, the language syntax allows for annotations that add arbitrary metadata to functions [13] (and since Python 3.6, to variables as well [14]). These annotations simply allow a user to associate some expressions with various parts of a function definition, they do not by themselves affect the semantics of the code. For variables (and function parameters), they are added by appending a colon followed by the annotation directly after the variable. For function return values, a "->" followed by the annotation is inserted between the parameter list and the colon that ends the function header.

Although practically any expression is usable as a function annotation, by far the most prominent use case of them became adding type information to function signatures. This practice was standardized in Python 3.5 [15], with clearly defined syntax and semantics of how these so-called type hints are to be written. We can see an example of this syntax in Listing 7.1. Notably, these are still completely optional and do not have a direct effect on the annotated code. They do not serve to restrict the runtime to be statically typed, they merely inform programmers and/or static analysis tools of expected types in the signature of a function.

Listing 7.1 Type Hints Syntax for Function Signature and Variables

```python
from typing import Union

def visit(edges: dict, start: str, visited: set[str]) -> set:
    visited.add(start)
    neighbors: Union[set[str], None] = edges.get(start)
    if neighbors is not None:
        for neighbor in new_neighbors:
            if neighbor not in visited:
                visit(edges, neighbor, visited)
    return visited
```

Since type hints are just metadata, nothing in the language itself enforces or ensures their correctness. Annotations such as `x: str = 3` are legal and will not cause the running program any difficulty, though there are external tools that can detect and warn against such errors.

Utilizing type hints is tricky for application code because we do not know how much it will be annotated, or how correct any provided annotations are. As a reference point, at the time of writing, no prospective customer of Manta interested in the Python scanner provided a single code sample that contained any type hints.

Libraries are a different story. The APIs of at least commonly used libraries tend to be well-annotated to inform developers on how to use them. If we were able to parse the type hints associated with return values of functions, we could use those to generate appropriate type flows whenever handling such functions. These annotations are present either directly in the source code of the library or in separate repositories containing *stub files*—files containing only the annotated signatures of classes and functions without implementations. Appendix B contains a list of places we could use to find type hints for each plugin currently in our scanner.

Tempting as this idea undoubtedly is, we found several issues that made it undesirable to implement.

7.1.1 Return Type Polymorphism

Suppose we have a class `A`, and a class `B` which is a subclass of `A`. If a function `foo` is annotated as returning `A`, it does not necessarily mean it returns only instances of class `A`; rather, it means that either `A` or any of its specializations (such as `B`) could be returned.

Any implementation relying on type hints for return values therefore must be able to traverse the class hierarchy not only upwards—for situations such as those described in Section 4.1.2—but also downwards, to determine all possible types returned from a function. In situations where these two problems combine, e.g.

when calling a method on an object returned from a library function, we would even have to do a combination of both, due to Python's multiple inheritance.

Note that we cannot even simplify this search by restricting ourselves to types that originate from the same library, as one might initially expect. Consider a scenario where a library function is passed a function object annotated as returning `A`, and then returns the result of calling that function (and is therefore also annotated as returning `A`). The actual return value can be defined in the scope of the caller, invisible to the library itself.

7.1.2 Hinting Complexity

After careful study, we must point out that the type hinting mechanism of Python is far more complex than initially thought. Following their initial introduction [15], no less than twenty-four subsequent PEPs[1] related to typing have been accepted, with three more open at the time of writing. Some of these extensions are fairly straightforward, such as simplifying the syntax for type unions [16] or adding a type for `Self`. These could be easily added to the scanner once initial support is established. Some, however, are much more problematic, and could even introduce completely new concepts to the analyzer such as structural typing [17], which is currently wholly out of scope for this work.

It must be said that this complexity need not be an issue by itself. We could initially support only the most relevant subset of these features, and add more as time goes on. Furthermore, it is our belief that outside of libraries most uses of type hints tend to stay fairly simple, if they are used at all.[2]

The real issue becomes apparent when examining the library stub files. In their desire to be both maximally correct and maximally expressive, the APIs of some libraries often create a lot of complicated, difficult-to-parse general types, which then permeate the entire library through importing, aliasing, subclassing, and combinations thereof. As such, it seems that a host of specific complex features would have to be supported before any meaningful percentage of these libraries was successfully parsed. Furthermore, the annotated source or the stub files can contain language features that our scanner either does not support or are not compatible with being analyzed symbolically. To name but one example, `if` statements that provide different function definitions based on which version of Python is used are common in API stubs.

[1]Python Enhancement Proposals.

[2]This was likely a contributing factor to our initial underestimation of the complexity of the type hint system.

Illustrative Example: `dict`

To see this complexity in action, let us look at a class as fundamental and (seemingly) simple as `dict`, the built-in dictionary. Its (abridged) stub definition[3] is shown in Listing 7.2.

Listing 7.2 Abridged API Stub of `dict`

```python
_KT = TypeVar("_KT")
_VT = TypeVar("_VT")

class dict(MutableMapping[_KT, _VT]):
    @overload
    def __init__(self) -> None: ...
    @overload
    def __init__(self: dict[str, _VT], **kwargs: _VT) -> None:
        ...
    @overload
    def __init__(self, __map: SupportsKeysAndGetItem[_KT, _VT]):
        ...
    @overload
    def __init__(self: dict[str, _VT],
        __iterable: Iterable[tuple[str, _VT]],
        **kwargs: _VT) -> None: ...
    def __new__(cls, *args: Any, **kwargs: Any) -> Self: ...
    def copy(self) -> dict[_KT, _VT]: ...
    def keys(self) -> dict_keys[_KT, _VT]: ...
    def values(self) -> dict_values[_KT, _VT]: ...
    def items(self) -> dict_items[_KT, _VT]: ...

    @overload  # type: ignore[override]
    def get(self, __key: _KT) -> _VT | None: ...
    @overload
    def get(self, __key: _KT, __default: _VT) -> _VT: ...
    @overload
    def get(self, __key: _KT, __default: _T) -> _VT | _T: ...
```

Just based on this partial definition of a single class (the actual `dict` stub is over twice as long), let us list the features we would need to support to be able to parse the class properly:

- The `@overload` decorator and overloaded methods, including overloads that have the same signature except for type hints.
- Generic classes and type variables, including partially specified generics.
- The `Self` return type.

[3]From `https://github.com/python/typeshed/`

- Type unions using the simplified syntax.
- The `Any` type.
- The `typing.MutableMapping` class, which inherits from the `Mapping` class, which inherits from the `Collection` class, which inherits from the `Iterable` class, the `Container` class, and the `Protocol` class. Each with its own set of methods to parse.
- Structural subtyping (due to the transitive inheritance from a class implementing `Protocol`).
- The `SupportsKeysAndGetItem` internal protocol.
- Matching the correct overload to an invocation based on the type hints of the arguments, including protocols.
- The `dict_keys`, `dict_values`, and `dict_items` internal generic classes, each with its own inheritance chain.

That is a fairly substantial list already, and we have barely scratched the surface. Even looking a bit deeper into just the `dict` class, we find such features as covariant and contravariant type variables (`dict` is covariant in its value type, but invariant in its key type), definitions that change based on the Python version, type aliases, and more.

For a more comprehensive overview of the claimed complexity, Appendix C gives an analysis of every approved PEP related to typing and the extent to which it would impact the implementation of this proposed parser.

7.1.3 Accuracy

As was already said, type hints are not enforced by the runtime. Libraries—especially reputable ones—are probably more vigilant than most in keeping everything correct, errors still happen, whether because of a mistake made by the developers or because of inaccuracies caused by our scanner.

7.2 Plugin Configuration

Not only did we initially underestimate the full intricacies of type hints, it seems we also somewhat overestimated the effort required to add propagation types manually. For context, Table 7.1 shows the total number of functions handled by each data flow plugin. With this in mind, we decided to extend the existing

Plugin	Number of Configured Functions
`builtin`	202
`pyspark`	63
`pandas`	11
`databricks`	9
`sqlalchemy`	6
`delta-spark`	5
`awsglue`	3
`boto3`	3
Total	**302**

Table 7.1 Configuration Size of Plugins For Different Libraries

configuration to also provide propagations for type analysis, then manually configure those propagations.

The configuration of data flow plugins is defined in XML files. Each plugin is encapsulated in a top-level `<Plugin/>` tag. Each module in the plugin has its own `<Module/>` tag, which stores the name of the module as an attribute and any classes or functions handled within that module as children. The most notable part of the configuration is the `<Function/>` tag, which describes the actual configuration of a single function. It naturally contains a `name` attribute, describing the function's name. As its children, it first has a list of `<Argument/>` tags, describing the number and names of parameters the function has. Then comes a list of `<Propagation/>` tags (see Listing 7.3 for an abridged example of such a configuration).

Listing 7.3 Abridged Plugin Configuration of `list`

```xml
<Module name="builtins.py">
  <Class name="list">
    <Function name="append">
      <Argument name="self"/>
      <Argument name="object"/>
      <Propagation mode="collectionAdd"
          from="arg(object)" to="receiver"/>
    </Function>
    <Function name="pop">
      <Argument name="self">
      <Argument name="index">
      <Propagation mode="getItem"
          from="receiver,arg(index)" to="returnValue"/>
    </Function>
  </Class>
</Module>
```

A <Propagation/> tag defines a single propagation that should happen when a function is processed. It does so using three attributes. The first two, from and to, specify the source and target endpoints of this propagation and can have several values. An endpoint in the format of "arg(name)" refers to the parameter of the function named name. For instance methods, a "receiver" value can be used as a shorthand for "arg(self)". A "returnValue" target means the result of the propagation is assigned to the return value of the function. Finally, specifying "context(name)" lets us reference a special ephemeral context variable that only exists while processing the function. This endpoint allows us to store intermediate values and chain different propagations together into the final result. The endpoints need not have just a single such value, they can contain a comma-separated list of them.

The third attribute is mode. The handler for each propagation mode is implemented as its own class that encapsulates its name and internal logic. When processing a function, the value of the mode attribute is used to find and invoke the appropriate handler with flows from places specified in the from endpoint. The flows resulting from this invocation are assigned according to the specified target endpoint. The invocation also receives an instance of the PropagationContext class, which stores metadata about the invocation such as the caller context and the executable summary of the processed function. It is also used to store the current context endpoint data.

The list of propagations is optional. If a function is configured without having any <Propagation/> tags, it is taken as an indicator that no interesting flows are created by it. Such a function is marked as *skipped* and is not handled at all when encountered.

7.2.1 Type Analysis Extension

This configuration system is surprisingly versatile, and because we are using flows to propagate type information, only a few changes were needed to add support for type analysis. First, a new <TypePropagation/> tag was added. This tag has the exact same syntax as <Propagation/>, with the only difference being that one is only considered during type analysis and the other only during symbolic analysis.

Second, we created a new source endpoint marked as "type(name)". This endpoint is used when the source of a propagation is a predetermined type. For example, a function always returning a string may be configured with a type propagation passing flows directly from type(str) to returnValue. For situations where we know we cannot figure out the returned type, a special type(#Undetermined) variant of the endpoint may be used, which returns UndeterminedTypeFlow.

This configuration of course has to be manually added to each plugin. As part of this work, we have added the configuration for the `builtin` plugin where appropriate. The configuration of the other seven plugins mentioned in Table 7.1 is left as future work, though we feel there should be no major obstacles to their addition using the current system.

Third, new propagation modes had to be added to implement the specifics of type propagation. We introduced eight new modes, which we can group into three conceptual categories.

Initializers

The `collectionType` propagation mode takes two source endpoints: a collection type and an inner type. It wraps each flow from the inner type endpoint in a `CollectionTypeFlow` with the collection type and propagates it to the target. The `setKeyValueType` propagation mode is similar, except it takes three source endpoints instead of two and creates `KeyValueTypeFlow` instances. The third mode in this category is `initializeCollection`, which is used for creating collections from other collections. It takes two source endpoints, then for every `CollectionTypeFlow` in the second source propagates a copy of that flow, except its collection type is changed to that of the first source endpoint.

Unwrappers

The `getCollectionValueTypes` takes all `CollectionTypeFlow` instances in the source and propagates their inner value type flows into the target. The `getCollectionKeyTypes` mode does the same with the key type flows of only `KeyValueTypeFlow` instances, and `getCollectionIterationTypes` unwraps the correct flow that would be returned when iterating the collection. Lastly, `getCollectionItemTypes` is used to emulate the behavior of `dict.items()`. It turns each `KeyValueTypeFlow` in the source into a two-tuple containing the key and value flows, then propagates a tuple of those two-tuples.

Superclasses

The final propagation is `getSuperclasses` and exists to emulate the call to `super()`. With no source endpoints required, it only looks at the class containing the called method, takes its superclasses, and propagates a `SimpleTypeFlow` of each of those superclasses to the target.

7.3 Plugin Composition

The additions to the configuration require support in the plugin handler and parser of plugin definitions. As with much of the design of type analysis, the goal is to be as transparent as possible to the inner components utilized by the analyzer, i.e. as few components as possible should care whether symbolic or type analysis is running. The implementation is a reflection of this general principle.

We should start with the simple observation that `<Propagation/>` and `<TypePropagation/>` elements never need to be accessed at the same time—they are each used only during one part of the analysis. Besides this differentiation, the tags have basically identical semantics. This means we need hardly any new constructs to create the parsed representation of a `<TypePropagation/>` tag. The classes used for parsing `<Propagation/>` suffice.

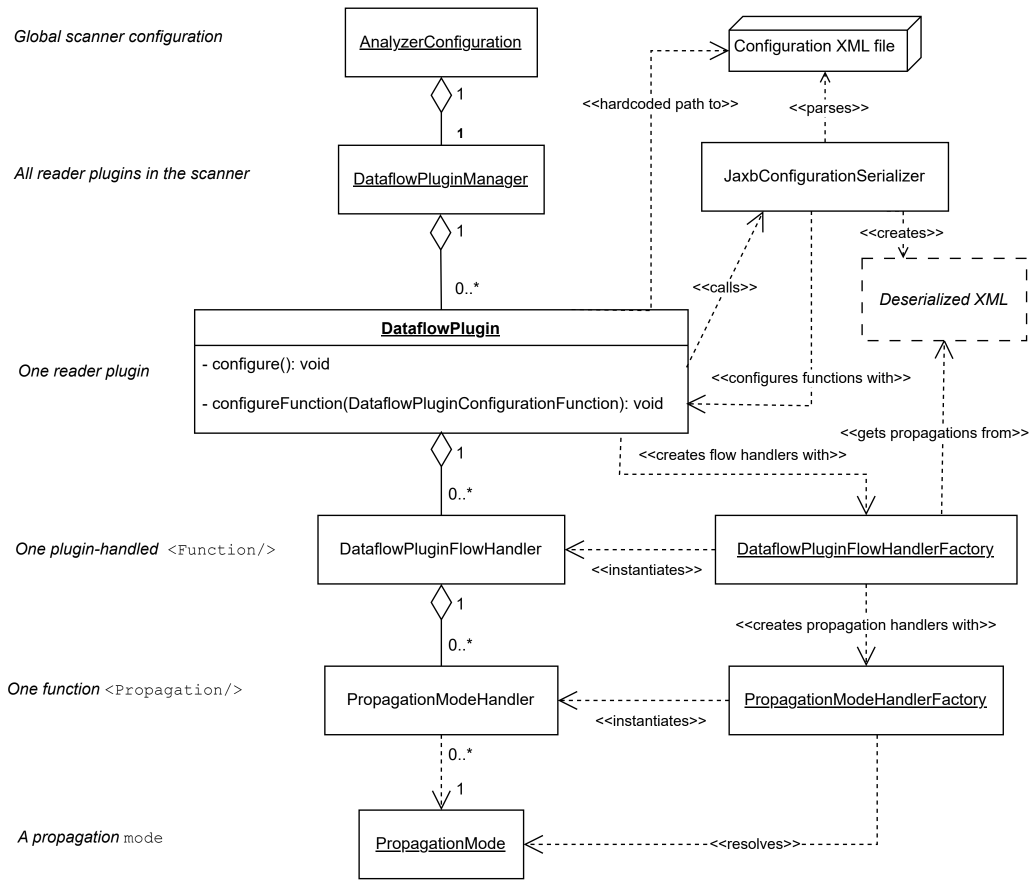
7.3.1 Parsing Plugins

The configuration of each plugin is parsed into an object of the `DataflowPlugin` class. Each instance of this class stores a map of all functions configured in that plugin and their associated handlers, which are represented by the `DataflowPluginHandler` class. Each handler contains a list of instances of the `PropagationModeHandler` class. These instances represent parsed handlers of individual `<Propagation/>` tags in the function, and they contain the resolved source and target endpoints as well as the `PropagationMode` object corresponding to the mode set on that tag (see Figure 7.1).

At the top of this hierarchy is the `DataflowPluginManager` class. The plugin manager is a container for all the plugins and is responsible for providing an interface the rest of the scanner can communicate with, as well as delegating any requests to the appropriate plugin. To that end, each plugin implements a `canHandle()` method that checks whether a given function is in its configuration, and the plugin manager then dispatches requests using the *chain of responsibility* pattern.

The hierarchy is composed using the Spring IoC container, with relevant classes configured as Spring beans and automatically injected with necessary dependencies (see Figure 7.2). Some parts are created directly, with dependencies passed into their constructor, others are made using dedicated factory methods or classes. The process of composing the final structure goes approximately as follows:

1. The beans for each propagation mode are created and registered into the bean for `PropagationModeHandlerFactory`.
2. The bean for `DataflowPluginFlowHandlerFactory` is created.



Global scanner configuration
AnalyzerConfiguration
1
1
All reader plugins in the scanner
DataflowPluginManager
1
0..*
One reader plugin
DataflowPlugin
- configure(): void
- configureFunction(DataflowPluginConfigurationFunction): void
1
0..*
One plugin-handled <Function/>
DataflowPluginFlowHandler
1
0..*
One function <Propagation/>
PropagationModeHandler
0..*
1
A propagation mode
PropagationMode
Configuration XML file
<<hardcoded path to>>
<<parses>>
JaxbConfigurationSerializer
<<creates>>
Deserialized XML
<<calls>>
<<configures functions with>>
<<gets propagations from>>
<<creates flow handlers with>>
DataflowPluginFlowHandlerFactory
<<instantiates>>
<<creates propagation handlers with>>
PropagationModeHandlerFactory
<<instantiates>>
<<resolves>>

Figure 7.1 Plugin manager construction. The center column shows the main constructed hierarchy, with the left column explaining its semantic equivalent and the right column containing other auxiliary classes. Underlined classes are created as Spring beans.

3. The beans for each data flow plugin are created. During that process,
   (a) The XML configuration of the plugin is deserialized.
   (b) Using a visitor that was passed to the deserializer, the plugin's `configureFunction()` method is called on each deserialized function.
   (c) This method in turn uses `DataflowPluginFlowHandlerFactory`. That factory extracts all deserialized propagation tags and uses `PropagationModeHandlerFactory` to create all propagation mode handlers.
   (d) The resolved propagation mode handlers for a single function are wrapped in a `DataflowPluginPropagationHandler`, which are then all stored in the plugin we are currently constructing.
4. With all reader plugins composed, the `DataflowPluginManager` bean is constructed.

7.3.2 Type Analysis Extension

As we described in Section 5.3.5, we can accomplish the support for type analysis plugins by injecting a different plugin manager into the data flow analyzer. To do that, we had to change the construction and composition of the plugins so that two managers are created instead of one. These plugin managers are practically identical in structure, except one is constructed using only `<Propagation/>` tags and the other using only `<TypePropagation/>` tags. This necessitated a few updates.

`DataflowPluginFlowHandlerFactory` used to be a regular class and got the list of propagations assigned to a function directly from the deserialized XML configuration. We turned it into an abstract class with an abstract `getPropagations()` method. It has two implementations, one for each kind of analysis, with each implementing that method to return the appropriate list of propagations. Since both tags have the same syntax (and virtually identical semantics), they can be represented by the same type once deserialized, so this approach is sound and the rest of the parsing code can work the same way regardless of which one is used.

Changes were also necessary in the Spring configuration. Each of the now two instances of the flow handler factory needs to be created in its own bean. Each of the newly created type analysis plugins also needs to be created in a bean. All plugins are represented using `DataflowPlugin`, the only difference between those used in symbolic analysis and those used in type analysis is which flow

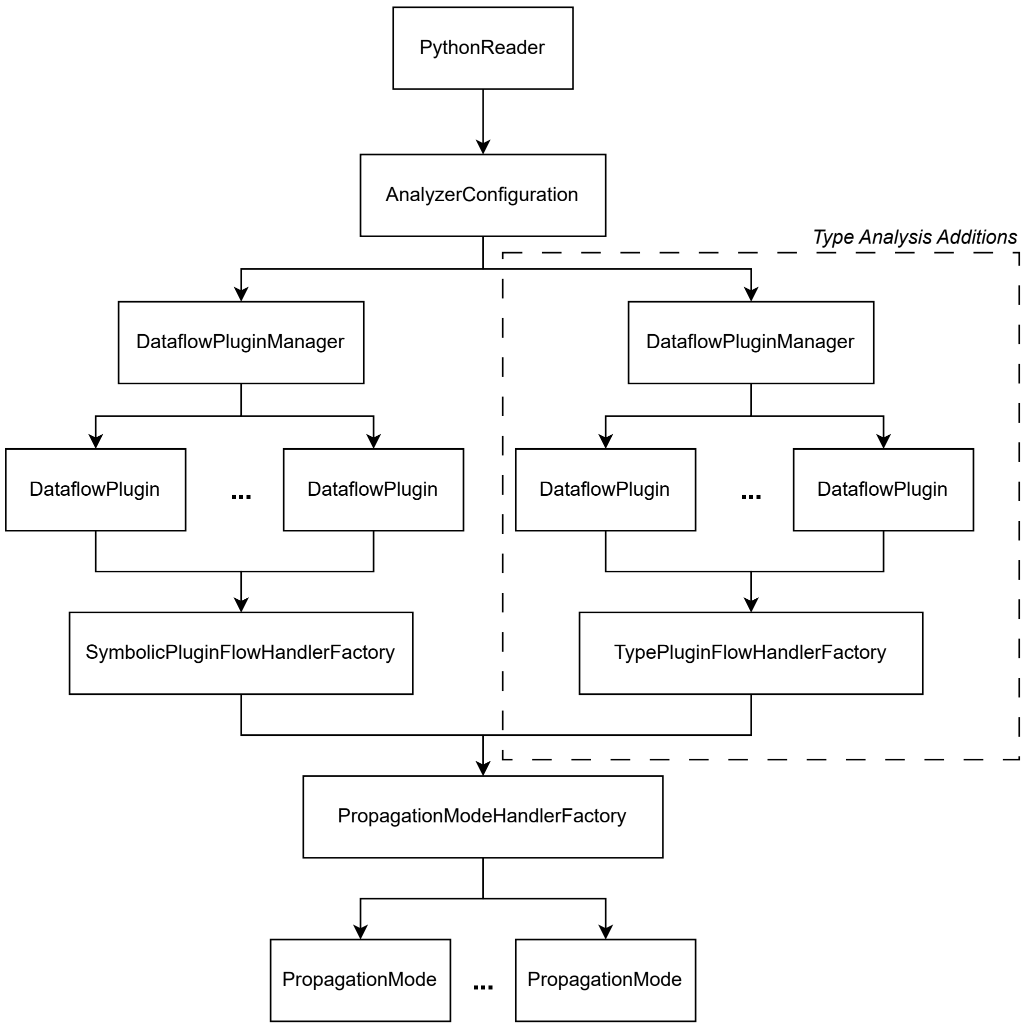


Figure 7.2 Structure of the Spring configuration. An arrow from bean A to bean B represents “A is dependent on B”.

handler factory is injected into them. To be able to differentiate the two when composing the plugin managers, a qualifier[4] of either `"symbolicAnalysis"` or `"typeAnalysis"` was added to each plugin bean.

With those changes made, we can parse the configuration into two separate plugin managers that are appropriately passed into the data flow analyzer.

7.4 Propagation Context

There was still one issue left to address, and it was related to the propagation context. We can recall from Section 7.2 that the propagation handler is passed an instance of the `PropagationContext` class with metadata about the analyzed invocation. We needed to add fields relevant to type analysis into this context. Specifically, the `type()` source endpoint needs access to the call graph and the `getSuperclasses` propagation mode needs access to the class hierarchy. Conversely, there are some fields in the context that type analysis either should not care about or might not even be available during type analysis.

To remedy this, the `PropagationContext` class was turned into an abstract class, containing only the fields relevant to both analyses, and two subclasses were created for it, containing those fields that weren't. This meant that most of the propagation mode handlers—children of the `PropagationMode` class—now had to depend on one specific `PropagationContext` subclass rather than on the generic version. That led us to implement two abstract subclasses of `PropagationMode` that cast the provided context into one of its implementations. The actual propagation modes were changed to inherit from one of those subclasses and use its corresponding propagation context subclass. Only a few modes that can be feasibly used in both variants were kept depending on the generic versions.

Once that was settled, we still had to find a way to create instances of the appropriate propagation context. These instances used to be created in the plugin manager and then passed down into the individual plugins, but since the plugin manager should be independent of the type of analysis it is used for, that is no longer an option. Instead, we lifted the creation outside of the plugin manager and made it accept an already-created context.

The plugin manager implemented a general interface for classes able to process executables called `IExecutableHandler`. This is the interface implemented by `ExecutableHandler` (responsible for processing application functions), `IdentityHandler`, and other auxiliary handlers. It is also the interface used to access the plugin manager from the method invoke manager. However, that interface naturally does not include any dependency on `PropagationContext`,

[4]`https://docs.spring.io/spring-framework/reference/core/beans/annotation-config/autowired-qualifiers.html`

so it does not make sense to implement once we introduce that dependency to the plugin manager. Instead, we created two different adapters over the plugin manager. These adapters implement `IExecutableHandler` by taking the passed arguments, creating the desired propagation context from them, and then delegating the call to the plugin manager. These adapters are what is actually passed into the data flow analyzer instead of the plugin managers themselves.

Chapter 8

Replacing the Call Graph

One of the main motivators for implementing type analysis in the scanner was to provide a better, more precise way to resolve targets of function invocations. In this chapter, we show how that was accomplished.

There are mentions of "statically resolvable" expressions throughout the chapter. In this context, that means expressions we can properly trace to the function/class/module they are referencing without needing to know the values of any variables or the results of any function calls. For example, if the statement `import m` exists in a module and module `m` contains a class `C`, then `m.C` is statically resolvable. The same is true for the expression `n.C` used under the statement `import m as n` and for `C` under the statement `from m import C`. Conversely, if we later make a variable assignment such as `x = m`, or define a function `foo` that returns the module `m`, then `x.C` or `foo().c` are not statically resolvable.

In practical terms, a statically resolvable expression is one that we can properly determine using only the results of infrastructure analysis.

8.1 Class Hierarchy

Suppose we know `x` is of type `A`. When considering the expression `x.foo()`, the definition of `foo` can be inside `A`, but it can also be inside any of its superclasses, or transitively in any of their superclasses. As a consequence, a properly constructed class hierarchy is a necessary part of any type-based call graph. The scanner had no preexisting mechanism to parse the class hierarchy of the analyzed application, so we had to create one.

8.1.1 Finding Superclass Descriptors

Superclasses are declared using a comma-separated argument list in the header of the class definition. To parse the class hierarchy, we need a way to map the expressions that define these superclasses to the classes they correspond to.

In this work, we limit ourselves to statically resolvable superclass expressions. As we have shown in Section 4.1.2, any expressions can be used as long as they resolve to a class object at the moment they are interpreted. However, trying to analyze them statically in their most general form is impossible, and most superclass expressions tend to be limited to straight names of classes, specified either directly or through an imported module, so this approach should be sufficient to cover most use cases. There are also ways to modify the superclasses of a class after it has been defined, for example by accessing the `__bases__` property of the class object. Such operations are also explicitly not supported.

Statically resolvable expressions come in one of two forms. Either a simple identifier (`x`), or a chain of identifiers (`x.y.z.C`). These can conceptually be thought of as the same thing, with a simple identifier being a chain of length one. To find the (statically resolvable) targets of such an expression, we mimic the scoping rules of the language.

First, we find possible targets of the first identifier in the chain (see Listing 8.1). We do this by starting at the current scope (the scope in which the class is defined), then moving upwards through enclosing scopes until we reach the module. In each scope, we look for possible executables that are defined in that scope under that identifier. We collect the descriptors of those executables into our starting set.

Listing 8.1 Finding the Start of an Identifier Chain

```
Set<IExecutableDescriptor> findExpressionBase(
    IExecutableDescriptor scope, String identifier) {
    var targets = new HashSet<IExecutableDescriptor>();
    // Walk up the enclosing scopes, searching each of them
    while (parent.getDescriptorType() != MODULE) {
        targets.add(scope.findIdentifier(identifier));
        scope = scope.getParentDescriptor();
    }
    targets.addAll(scope.findIdentifier(identifier));

    // Search the builtins module
    targets.addAll(builtinsModule.findIdentifier(identifier);
    return targets;
}
```

Luckily, the descriptor map provides us with the `findIdentifier()` method on each descriptor that allows us to search for executables found in that descriptor under the provided name. This method handles class and function definitions as well as import statements, including import aliases and star imports, greatly simplifying this process for us.

One thing we must not forget is that the built-in namespace is always available, so we must also look in the `builtins` module for the start of the chain. In `str.count`, the `str` identifier is part of `builtins`.

We should also consider whether it is necessary to always walk through all the enclosing scopes. In real execution, the innermost scope that contains the desired identifier is always used. However, definitions and imports can be conditional, so we cannot rely on them always existing at runtime. For this reason, we have to take the safer option of always going all the way to the level of the module.

Once we have found the base of the chain this way, we traverse iteratively through the rest of the identifiers (see Listing 8.2). Taking the starting set from the previous iteration, we try to find executables under the current identifier in each executable from the set. This results in the starting set for the next iteration. We then repeat this process until we have gone through the entire identifier chain.

Listing 8.2 Updating the Set of Found Descriptors

```
identifiers.remove(0); // remove the head of the chain

// iterate through the other identifiers in it
for (String identifier : identifiers) {
    targets = targets.stream()
        .flatMap(descriptor -> descriptor
                    .findIdentifier(identifier).stream())
        .collect(Collectors.toSet());
}

// remove non-class results
targets = targets.stream()
    .filter(desc -> desc.getDescriptorType() == CLASS)
    .collect(Collectors.toSet());
```

Since—by definition—only classes can be superclasses, the resulting set is filtered to only the class descriptors contained within, and those descriptors are the final result of the search.

Once we do this for all expressions in the argument list of the class, we get a set of all its possible superclasses. Repeating this for all classes in the analyzed application (we do not need it for plugin-handled functions, as those have explicitly defined propagations), we get a map from every application class to a set of its possible superclasses. This map serves as our class hierarchy.

Example

To fully illustrate this process, let us consider an example of a Python program with several modules found in Listing 8.3. In this example, we will be searching for all possible superclasses of `MainClass`. The class only has one superclass, defined by the expression `base.Class.Subclass`, which we are trying to find the possible targets for. The first step is finding the "base" of this identifier chain, in this case aptly named `base`.

We start with the scope in which `MainClass` is defined, which is the scope of the function `foo`. There is one dependency with the name `base` found in that scope, specified by an import statement. We add it to the set of possible targets. Then we move on to the parent descriptor, which is that of the `main` module. This module contains another dependency with the name `base` created by an import statement, as well as a class of that name. Both of these are added to our set. Since there are no more parents to go to from the module, we stop the search there. There is no class or function with the name `base` in the built-in namespace, so they do not contribute in this case.

```
targets == {main.base, module, package}
```

Next, we go through the rest of the expression segment by segment, trying to look for nested identifiers within found descriptors.

The first segment considered is `Class`, so we iterate through the three targets found so far to find a matching descriptor within them. The `main.base` class contains a `Class` class, so we add its descriptor to the new targets. The module `module` contains three possible matches for this identifier: two classes and one import dependency. All three are also added. The package `package` contains the module `Class.py`, which is accessed under the name `Class` from it, so we also add it to our set. Thus concludes the first iteration. To disambiguate between the targets, we will refer to some of them using the aliases added in the comments next to their definition.

```
targets == {main.base.Class, classes.Something,
    module.Class_1, module.Class_2, package.Class}
```

We move on to search for the next segment, `Subclass`. There is an executable of that name in `main.base.Class`, so we mark it as a possible target. The same is true for `classes.Something`, `module.Class_2` and `package.Class`. Notice that `module.Class_1` has no `Subclass`, so we simply drop that path from the list of possibilities. Thus concludes the second iteration.

```
targets == {Subclass_1, Subclass_3, Subclass_4, Subclass_5}
```

Listing 8.3 Superclass Resolution Example

```python
## main.py ##
import module as base

def foo():
  import package as base
  class MainClass(base.Class.Subclass):
    ...
  return MainClass()

class base:
  class Class:
    def Subclass(self):  # Subclass_1
      ...

class Class:
  class Subclass:        # Subclass_2
    ...


## module.py ##
from classes import Something as Class

if False:
  class Class:           # module.Class_1
    ...
else:
  class Class:           # module.Class_2
    class Subclass:      # Subclass_3
      ...

## package/Class.py ##
class Subclass:          # Subclass_4
  ...

## classes.py ##
class Something:
  class Subclass:        # Subclass_5
    ...
```

This is the final iteration, as we have reached the end of the superclass expression. As the final step, we remove all results that are not classes. This leaves us with `Subclass_3`, `Subclass_4`, and `Subclass_5` as the final possible targets for the expression.

Careful examination of the source reveals that these are in fact exactly the valid possibilities for that expression. We discarded `Subclass_1` due to being a function, and `Subclass_2` was never even considered, as it is not inside any valid `base`.

Imprecision

Let us revisit the limitation of handling only statically resolvable expressions. Even when we cannot properly resolve a superclass expression, there are situations when we can at least detect that we cannot resolve it. A typical case of this is when the expression itself is in an incompatible form, such as when a function call or an unpacking expression is encountered.

To note such situations, the values in the class hierarchy contain not only the set of possible superclasses but also a boolean flag that identifies whether there were any issues during parsing. This flag is set as true if either the expression is in a form that is not statically resolvable, or the final set of targets is empty. That way, subsequent analysis does not have to rely on the possibly erroneous assumption that all superclasses were resolved correctly and are included in the class hierarchy.

Nested Packages

So far we have considered traversing through the chain of identifiers segment by segment. That may not necessarily be semantically correct. Consider the following snippet, where `p` is a package and `m` is a module within it.

```
import p.m
class B(p.m.A): ...
```

Our solution would first search for a descriptor corresponding to `p`, then for `m` within that descriptor, and finally for `A` within that. However, `p` does not correspond to a module, and no dependency is imported as `p` into our scope. Luckily, infrastructure analysis handles this for us.

When we fetch the identifiers represented by an import statement such as the one above, we not only get a reference to `p.m`, but also a faux reference to `p`. If `p` is a regular package, the value under `p` in this mapping is the descriptor for the `p.__init__` module whose dependencies are artificially extended with all modules present in the package. If `p` is just a namespace package, a special

#namespace_init descriptor is created for it, also containing the modules in that namespace.

8.1.2 Searching for Functions in Superclasses

Once we have the class hierarchy constructed, we can query it to find functions defined within the ancestry of a given class.

In Python, this search uses the C3 linearization order, as shown in Section 4.1.2. Sometimes that search stops after finding the first match (e.g. when finding identifiers), sometimes it goes through the whole list (e.g. when handling super() calls).

We could try replicating that process, but the possibility of conditional definitions makes that challenging. If we stop the search at some class C that contains the desired method, there is no guarantee that that class was actually created during any particular execution, or that the method was actually created within it. There is also the possibility that C is in the hierarchy only as the result of some over-approximation, meaning stopping there would never be correct. We are left with no better option than to always walk through the whole ancestry of the class.

This walk presents another potential pitfall. Hierarchies in Python have to be acyclic as a result of the limitations imposed by the linearization method, which means the ancestry of a class always forms a DAG. No matter how one might try to define or modify the list of superclasses, if it cannot be linearized, the operation results in an exception being thrown by the interpreter. Just because cycles cannot appear at runtime, however, does not mean they will not show up in our constructed hierarchy. Consider the example in Listing 8.4.

If condition always evaluates the same for both files, then this is valid Python that will execute without error. Either B inherits from A in the if branches, or A inherits from B in the else branches (see Figure 8.1).

We cannot know which is which, nor can we know anything about the equality of the two conditions, so we have to consider all possible choices. When analyzing C, we have to link to both definitions of A as parents. The second A definition is linked to both definitions of B for the same reason, as is the first definition of B to both definitions of A. Those last two statements cause a cycle in our parsed hierarchy between the second definition of A and the first definition of B, where each refers to the other as its ancestor.

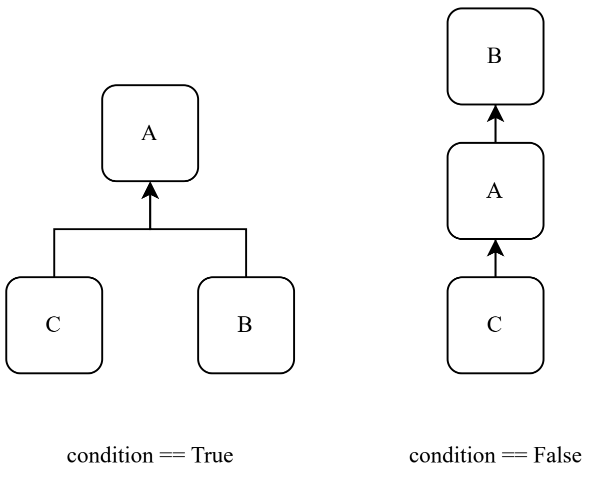


Figure 8.1 Possible Hierarchies in Listing 8.4

Listing 8.4 Cycles in the Hierarchy

```python
## a.py ##
if condition:
  class A: ...
else:
  from b import B
  class A(B): ...

## b.py ##
if condition:
  from a import A
  class B(A): ...
else:
  class B: ...

## c.py ##
import a
class C(a.A): ...
```

While we have to be mindful of such situations, they do not pose a significant challenge to the algorithm at large. We only have to make sure we are searching the ancestors in a way that can detect cycles. We chose to implement it using a simple DFS search, which handles them natively.

As we are searching, we keep track of whether any of the found superclass sets have the “contains unresolvable expression” flag set. If we find one that does, we immediately abort the whole search and return an empty result. We do this because in such cases we know that whatever ancestry we would return would not be fully correct. Rather than do that and risk a potential loss of targets, we let the caller know that the search failed and it can resort to a different approach that does not rely on the hierarchy.

8.2 Building a Call Graph

Throughout this thesis, we have shown how difficult it is to create anything resembling a static call graph for a Python application. Now, we show how we use the information provided by type analysis to build one that, while not perfect, is a definite improvement on the existing mechanism.

This process occurs, perhaps unsurprisingly, in the execution visitor for type analysis. Earlier in Chapter 6 we alluded to the idea that when handling invocation expressions, the visitor has to find the possible targets of the invocation. This search happens in several steps.

First, the expression is viewed as though it is statically resolvable, and all targets that are reachable as such are found. Any functions found this way are added to the list of possibilities directly. For any classes found this way, we add their constructor, if they have one.

Next, if the function is called on some object, we use a second search, based on collected type flows. This search starts by getting the flows of the object on which the function is called (`a.b` for `a.b.foo()`) and getting the types those flows represent. For each of those types, we look inside it, and inside any classes in its ancestry, and try to find methods with a matching name. Any found are added to the list of possibilities.

Finally, the list of possibilities is filtered to only those functions whose signature can match the arguments provided in the invocation. This final collection is then stored as the possible targets for that expression.

The visitor does all of this using one of its dependencies, a class named `TypeAwareInvocationResolver`. This class is responsible for performing the aforementioned searches and for storing the possible targets for invocations. It is one of the outputs of type analysis and serves as the improved call graph used in symbolic analysis. During symbolic analysis, a search of possible targets for an invocation expression translates to a lookup in the table of saved targets that were found during type analysis.

8.2.1 Iterative Construction

This method looks deceptively simple, but there are some questions that have to be carefully considered. One such question concerns the correctness of the method. The second search relies on the type flows found during analysis so far, but those flows may not necessarily be complete. Can we be sure that this partial result will not cause us to miss any possible targets?

It turns out we can, thanks to our usage of the worklist algorithm. When encountering a function call for the first time, if we do not know all possible types of the object it is called on, we can use only the ones we know right now, possibly skipping the function if no types are known. If more possible types are discovered later, they will be processed in a subsequent iteration of the worklist.

8.2.2 Over-approximation Safeguards

We also have to ensure that we detect situations when the search is meaningless or imprecise and respond accordingly. We included several checks in the search process for this purpose.

If there is an `UndeterminedTypeFlow` in the flow set of the object on which a function is called, we abort the type-based search before even starting it. If the ancestry of one of the types contains an unresolvable superclass expression, the whole search fails. If either of these happens, the expression is marked as "unknown" in the call graph and we fall back to the old imprecise method of searching for targets.

Note that we cannot do the same if no targets were found. We might imagine that an empty result is caused by an error or unsupported feature, but that need not be the case. Because we are creating the call graph iteratively, it could very well be the case that the correct targets will be found in some future iteration, and prematurely marking the expression unknown would ruin our chance to do so. If, however, no targets for an invocation were found during the whole type analysis, we can safely treat them as unknown.

The search for targets made during symbolic analysis takes these safeguards into account. If an expression is marked as unknown or has no associated targets, the fallback search is used. Otherwise, we return the targets stored in the map of our call graph.

Chapter 9

Integration

This chapter describes other changes made as part of this work that do not necessarily relate to its main goals but were necessary or desirable for our purposes.

9.1 Removing the Old Call Graph

Recall from Chapter 2 that infrastructure analysis occurs in two parts. The first creates the descriptor map, which tracks import dependencies and hierarchical relationships between executables. The descriptor map is also responsible for the old way of finding invocation targets.

The second step creates the call graph, which stores caller–callee relationships between executables. These relationships are then used when administering the worklist. At least, that was the initial idea and what all available documentation described at the time of writing.

As we found out during our work, this explanation was not accurate to the actual implementation and hadn't been for over two years. While the call graph was being constructed (by finding the possible targets of all invocations found), the resulting mappings were not used anywhere in the scanner. Instead, symbolic analysis creates its own call graph, which is built iteratively—whenever the execution visitor processes an invocation, it registers the caller and callee invocation contexts. The call graph created by infrastructure analysis was only used for delegating calls that were handled by the descriptor map.

We can only speculate why this change was made, as there does not appear to be any documentation related to it and the person who made it no longer works at the company. Our best theory is that the main reason was a difference in granularity. The extendable call graph created by symbolic analysis can operate directly on invocation contexts, which makes it easy to determine which contexts should be added to the worklist after one context is processed. By contrast, the infrastructure call graph worked with executables. Relying on it to update the worklist would therefore result in both more work and possible imprecision, as adding the caller/callee of an executable would mean adding all contexts in which that executable appears.

As part of our work, we removed the infrastructure call graph—as well as its construction, associated interface, and related classes—from the scanner and updated its usages to depend directly on the descriptor map. An appropriate change was of course made to the documentation where required.

9.2 Reworking Group-Handled Functions

In Section 6.1.4, we talked about the processing modes assigned to each function to determine how the method invoke manager deals with them. Aside from the four modes mentioned there, the scanner supports a fifth mode, *group-handled*.

9.2.1 The Idea of Group Handling

There are functions that tend to behave mostly the same in any class. This is typically true of Python's *magic functions*—functions that represent some operator. A call to `__eq__()` will pretty much always just return a boolean value, and calls to `__next__()` tend to return individual elements of the underlying iterator. Many of these functions don't even produce data that is important for data lineage, so they can be skipped entirely. As these methods tend to be both omnipresent and similar in function, manually configuring them for each class in each plugin becomes extremely tedious. However, unannotated functions become *identity-handled* by default, which can introduce a lot of imprecision into the flow results.

This is where the concept of group handling comes in. We define one particular function as *group-handled* and configure it with the desired propagations. This function then serves as a prototype of that "group" of functions. If an invocation target is ever found that has the same signature as this prototype and would be identity handled, it is instead handled the same way as the prototype.

Any functions that work differently from the group prototype can of course be overridden. By explicitly configuring a function, we effectively remove it from any group it would fall into and let it be handled completely separately.

It is worth noting that this concept is, at its essence, a convenience. The plugin system (and the scanner as a whole) could in theory very well work without any group-handled functions. However, the sheer number of entries that would have to be manually added to each plugin—many of them repeated—make it a practical necessity. There is also one other slight benefit it gives us; if multiple functions in the same group are found, we can use the propagations of only one of them, since we know the others are handled identically, saving some execution time.

9.2.2 The Reality of Group Handling

The way these functions were declared in the scanner was, suffice it to say, a little hacky. We said that group-handled functions are determined based on their signature, but while that was the overall effect, it is not quite representative of how they were implemented.

Firstly, `FunctionProcessingModeResolver`, the component responsible for assigning processing modes to functions, had a hard-coded list pre-determined list of specific functions. The functions in that list represented the prototypes and were explicitly assigned as group handled. All others, including the aforementioned not configured ones, get their regular processing mode (*identity-handled* in those cases). Then, when resolving targets of an invocation, after all the possible targets were found, we would check whether at least one of them was assigned as group handled. If so, we would remove every identity-handled function from the result set (see Figure 9.1).

This process does not really reflect the described theoretical ideal of group handling. For example, we can see that it can only work if we guarantee that the prototype is in the set of possible results for each invocation that also includes another member of the group. The reason the implementation resulted in something that ended up mostly similar to the theory is due to a confluence of two factors.

The first is that every declared prototype was inside the `builtins` module. This is not a requirement that was imposed inherently by the system, but it was the only way to make things work. That is because the `builtins` module is reachable from everywhere, so it is always possible to reach its functions with any all. If a function from any other module were to be selected as the prototype, there would be a chance that the module it is contained in was not imported into the scope of the caller, in which case the function would not make its way into the list of possible invocation targets and the whole process would break.

The second factor is related to the first and has to do with the imprecision of how invocation targets used to be found. This imprecision practically ensured that whenever a not configured function with the same signature as the prototype was called, the prototype was also included in the possible results (because, once again, built-in functions are reachable from everywhere) and the prototype would cause all the not configured functions to be removed.

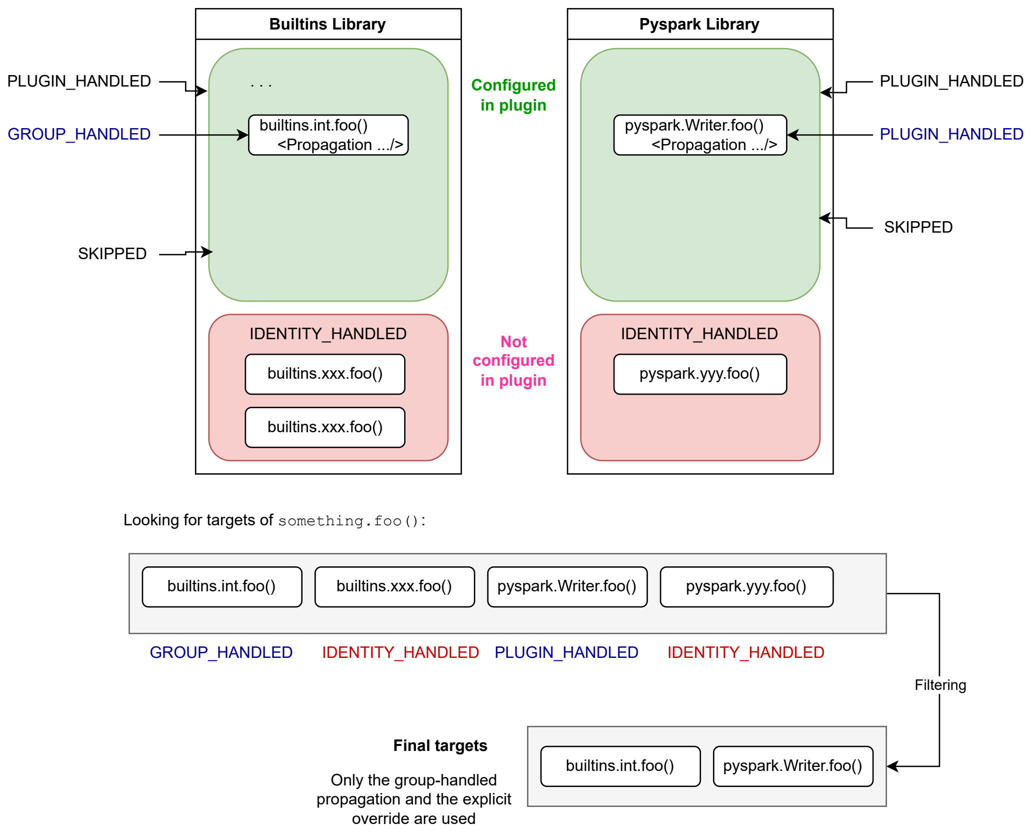


Figure 9.1 Old Plugin Configuration and Target Filtering

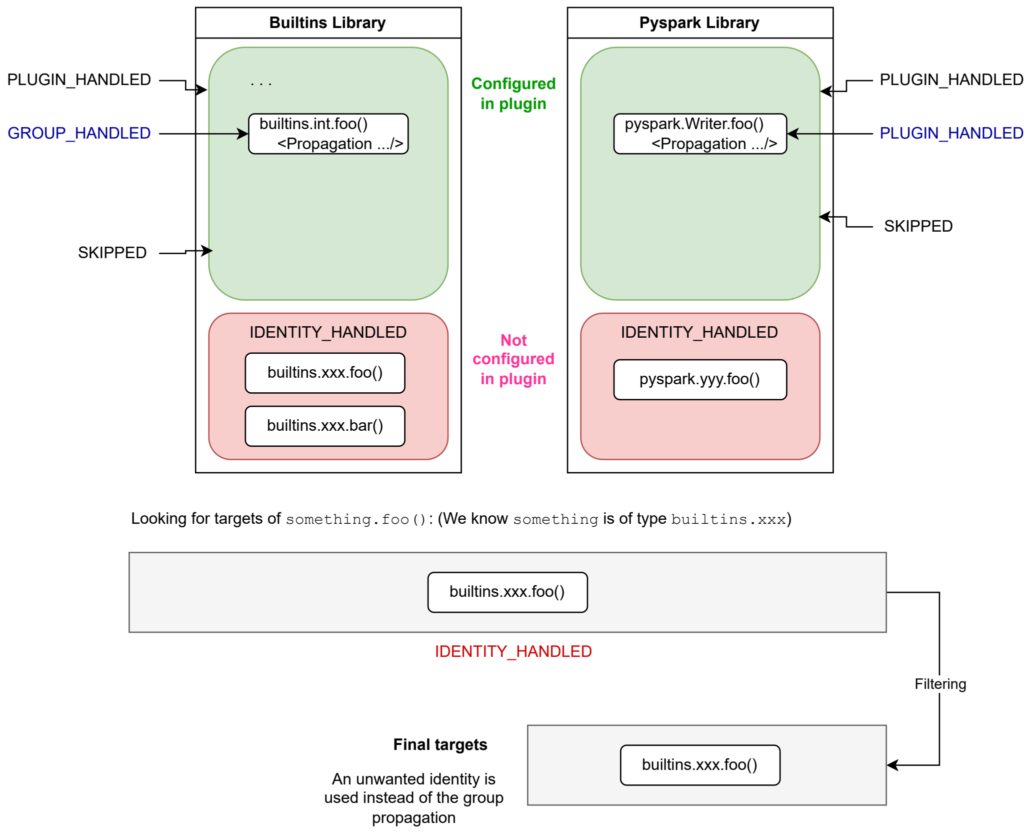


Figure 9.2 How Old Configuration Breaks with Type Analysis

9.2.3 Fixing Group Handling

We can see how the introduction of type analysis would cause a problem here. Once we make the search for invocation targets more precise, we can no longer operate under the assumption that any call to e.g. `__eq__()` will consider `#__builtins__.int.__eq__()` as a possible target (see Figure 9.2). As such, we had to completely change the way group-handled functions are created and utilized.

Our first observation was that we had to get rid of the reliance on prototypes, as any such system runs into the possibility that the prototype may not be included as a possible target. Instead, we created a separate new plugin specifically for group-handled functions. This plugin is configured and composed the same way as any other plugin, just with slightly different semantics (see Figure 9.3).

When asked whether it can handle a function descriptor, this plugin ignores the module and class in which the descriptor is found and looks only at its name and parameter list. If those match any function in the plugin's configuration, it can be handled by the plugin. This is a meaningful difference from all other plugins,

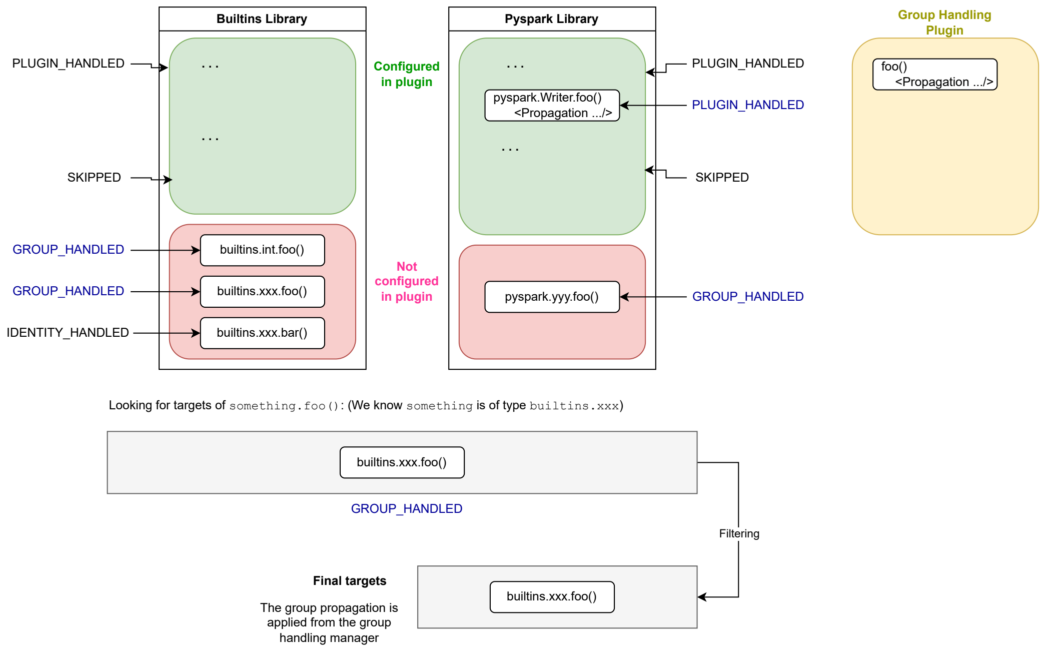


Figure 9.3 Updated Plugin Configuration

which only handle functions in the modules and classes they were configured for.

This plugin is injected as a special dependency into the plugin manager, outside of the list of other plugins used. When the plugin manager handles a function, it calls on this group-handling plugin only if no other plugin is able to process the request, thus ensuring it does not override an explicit configuration of some function.

Another change we made was to the assignment of processing modes. Whereas before only the prototype was assigned as *group handled* and all the other members of the group as *identity handled*, now we set every function in the group as *group handled*. We determine whether a function is in a group by the fact that it can be handled by the group handling plugin and has no explicit configuration elsewhere. We can still make the same optimization as before, except instead of keeping the group-handled prototype and throwing away identity-handled functions, we pick one random member of the group and throw away all the others.

9.3 Refactoring Function Processing Modes

Related to the last change, the way processing modes used to be assigned was somewhat problematic and caused issues for type analysis.

Type analysis creates its own set of plugins, which are similar but semantically distinct from those used in symbolic analysis. We may want to process plugin-handled functions differently in type analysis and symbolic analysis. For example, a function for which it is too difficult (or nonsensical) to define a propagation (and is therefore marked as skipped) can have a well-defined type which we can use in the type analysis plugin. However, since the assignment of processing modes happens during infrastructure analysis, it used to be unable to discriminate between the two kinds of plugins and blindly followed the rules defined for symbolic analysis plugins. This caused two problems:

1. A function that had no symbolic propagations configured but had some type propagations was considered *skipped* in both analyses, completely ignoring those type propagations.

2. A function that had no type propagations defined was either *skipped* or *plugin-handled* with no propagations—effectively also skipped. As mentioned in Chapter 7, we cannot afford to have functions without configuration produce no result. Instead, it must return an `UndeterminedTypeFlow`.

The first change we made was to extend each function descriptor to contain two fields for processing modes instead of one. A corresponding change was made to the resolver of processing modes, which now contains the necessary logic to assign both modes to each descriptor.

This change meant we also had to update the way these modes are used. Their main usage is in the method invoke manager, which branches on them to determine the correct way to handle a function. We want this manager to be analysis-agnostic, so we had to find a way to pass the correct processing mode to it. Luckily, there is a good place to seamlessly apply the generalization.

Inside the manager, the descriptor—and, by extension, its processing mode—was accessed through the invocation context created for the specific invocation being handled. We can therefore refactor the invocation contexts to return the processing mode directly, bypassing the need to go through the function descriptor without any major changes to existing code. Invocation contexts are created in execution visitors, which are unique for each kind of analysis and therefore know which processing mode they should use.

Finally, some changes were made to the structure of the resolver of processing modes. That resolver needs to depend on the plugin manager (to determine which functions are plugin handled), but it could not do so directly due to issues of cyclic dependencies, so this dependency was satisfied in a strangely roundabout way.

After the plugin manager was created, it serialized all of its plugins into two maps—one for plugin-handled functions, one for skipped functions. The processing mode resolver took those maps and used them during assignment—if a function was in one of those maps, it was assigned the corresponding processing mode.[1].

With the introduction of a second plugin manager and the shift of group-handled functions into a self-contained plugin, this approach became untenable, so we decided to simplify it. Now, each plugin manager implements the newly created `IProcessingModeFetcher` interface. This interface contains one method, `tryGetProcessingMode()`, which returns the determined processing mode of a given descriptor if able. The implementation in the plugin manager works using the following rules:

- If the function is handled by any regular plugin, return *plugin-handled*.
- Otherwise, if the function is skipped in any regular plugin, return *skipped*.
- Otherwise, if the function can be handled by the group handling plugin, return *group-handled*.
- Otherwise, return `null`.

The processing mode resolver takes two instances of that interface as dependencies, which are supplied by the two plugin managers. It then utilizes them to resolve both processing modes for each descriptor. Lastly, if the processing mode for type analysis is declared as *skipped*, it is rewritten to be *identity-handled*, to reflect the difference in semantics.

[1]Remember that group-handled functions used to be explicitly hard-coded in the resolver.

Chapter 10

Evaluation

In this chapter, we evaluate the effects of our work on the scanner in terms of precision and performance. It must be said that the expected advantages gained by using type analysis are highly context-dependent: applications with a lot of identically named functions should experience a greater degree of difference than applications where each name is unique. This makes it tricky to objectively quantify any purported improvements.

10.1 Methodology

Due to the nature of this work, it was integrated into the scanner gradually and concurrently with other changes. That makes it challenging to isolate the full extent of our changes. For our comparative evaluation, we decided to use the latest version of the scanner in both passes, with one having type analysis enabled and the other disabled (meaning the descriptor map is used directly).

The testing was performed on a Windows 11 laptop with an Intel Core i7-1185G7 processor and 16 GB of memory.

10.2 Precision

We encountered several improvements in the precision of resulting data lineage graphs when testing our solution. We show two illustrative examples here.

10.2.1 Collection Iteration

Take the code in Listing 10.1. It defines a single function that takes a list of strings and creates a file for each item in that list, into which it writes the string “message”. This function is called with a list containing a single item, `"foo.txt"`.

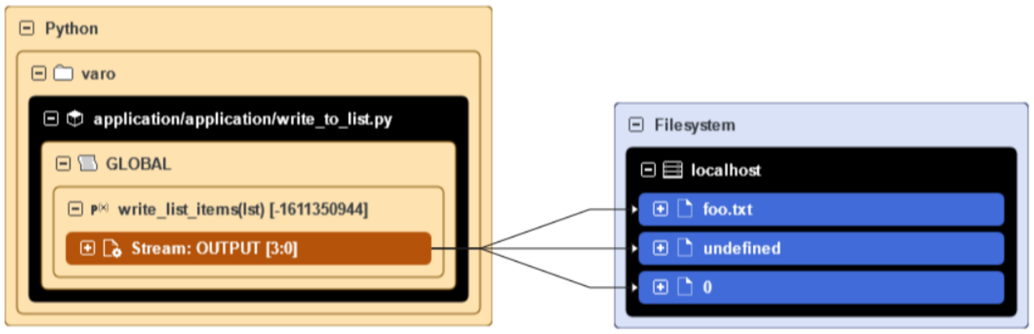

Python
varo
application/application/write_to_list.py
GLOBAL
write_list_items(lst) [-1611350944]
Stream: OUTPUT [3:0]
Filesystem
localhost
foo.txt
undefined
0

Figure 10.1 Lineage of Listing 10.1 with Type Analysis Disabled

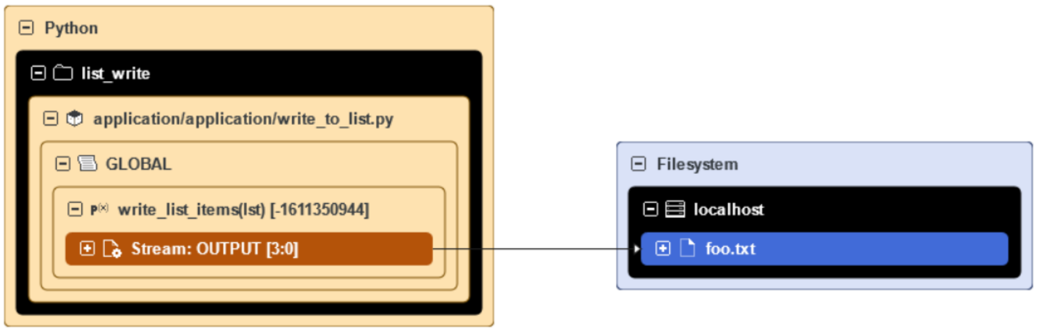

Python
list_write
application/application/write_to_list.py
GLOBAL
write_list_items(lst) [-1611350944]
Stream: OUTPUT [3:0]
Filesystem
localhost
foo.txt

Figure 10.2 Lineage of Listing 10.1 with Type Analysis Enabled

Listing 10.1 Writing to List Items

```python
def write_to_list_items(lst):
    for item in lst:
        with open(item, "w") as f:
            f.write("message")

lst = ["foo.txt"]
write_to_list_items(lst)
```

We would expect the data lineage of this code to consist of a single line with two ends, leading from the Python script (as that is from where the written contents originate) to the `foo.txt` file. Unfortunately, without type analysis enabled, we also find other flows in the output (see Figure 10.1).

The `0` output in particular is easily explained as the result of an over-approximation. When unwrapping the items of the list in the `for` loop, the propagation used for iterating over `list` is used to pass the list items into the `item` variable. However, the propagation used for iterating over `dict` is also used, passing the keys of the list into the variable as well (since `"foo.txt"` is at index 0 in the list).

After applying type analysis, the scanner is able to determine that the parameter is indeed a list and these extraneous flows disappear, as seen in Figure 10.2. Given that iteration is one of the most basic operations used in any application, we can imagine the cumulative effects of this improvement can add up over a larger application.

10.2.2 Separation of Instances

There was a long-standing issue where flows of different classes with identically named methods could end up getting mixed up together. A minimal reproducible example demonstrating this behavior can be seen in Listing 10.2.

Listing 10.2 Separation of Class Instances

```python
class A:
    def __init__(self):
        self.file = "a.txt"

    def call(self):
        pass

class B:
    def __init__(self):
        self.file = "file.txt"

    def call(self):
        with open(self.file, 'r') as f:
            print(f.read())

a = A()
b = B()
a.call()
b.call()
```

In the example above, the instance of `A` does not create any data flows, as its `call()` method is empty. The call made on an instance of `B` creates a flow from the `file.txt` file to the standard output. Unfortunately, due to an outstanding bug that was never properly diagnosed, the `"a.txt"` string somehow managed to get into the flow set of `b.file` at some point during the analysis (and vice versa), which resulted in the creation of a second flow from that file to the standard output (see Figure 10.3).

While we cannot precisely point to the root cause of this issue, we can say that improving the precision of locating invoked functions by enabling type analysis caused it to disappear, isolating each file path to its respective class (see Figure 10.4).

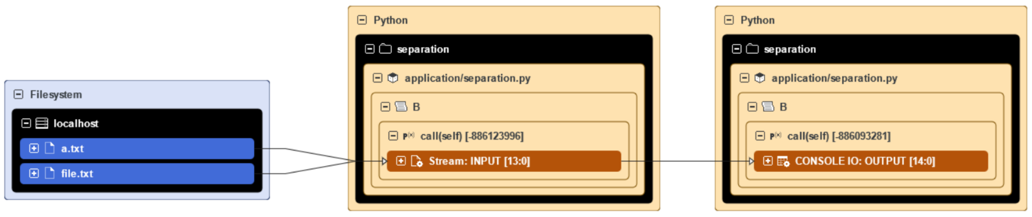


Figure 10.3 Lineage of Listing 10.2 with Type Analysis Disabled

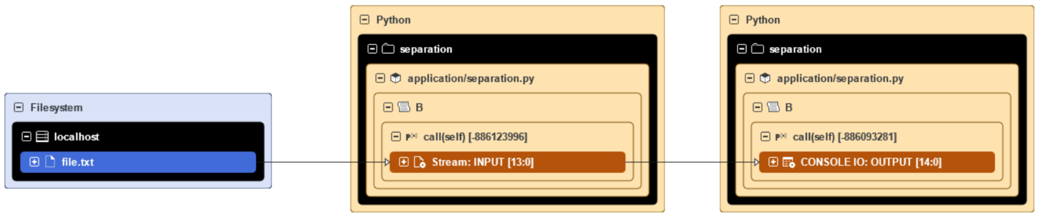


Figure 10.4 Lineage of Listing 10.2 with Type Analysis Enabled

10.3 Performance

We must stress that performance improvements were not the primary goal of this thesis. It was always more focused on improving precision and allowing simplifications in the rest of the scanner. To evaluate the performance of type analysis on a representative sample that is not biased toward its strengths, we chose to use a script provided to us by a prospective customer of the scanner. Because of this, we cannot share its source, but Table 10.1 describes some of its main characteristics. It uses the `awsglue`, `boto3`, `pandas`, and `pyspark` libraries.

Table 10.2 shows the time spent on each step of the scanner starting with infrastructure analysis for both passes, rounded to the nearest second. As we can see from these results, any computational cost incurred by the addition of type analysis is outweighed by the reduction of time later spent on symbolic analysis. This reduction was noticeable even though the source heavily relies on the aforementioned libraries, which are not yet configured for type analysis. We expect broadening this support in the future could increase the time spent on type analysis (as fewer functions become identity-handled), but might further improve the speed of symbolic analysis.

We cannot show the lineage generated from the source, due to both intellectual property rights concerns and deficiencies in the scanner that mean the produced lineage is not completely meaningful at this time, but we can reveal that the lineage remained unchanged in both versions, which we would expect, given once again the usage of libraries not configured for type analysis.

Lines of Code	471
Module Count	1
Function Count	11
Invocation Count	354
Number of Imported Modules	9
Number of Imported Classes	15
Number of Imported Functions	4

Table 10.1 Characteristics of the Evaluated Code

Step	**Without Type Analysis**	**With Type Analysis**
Infrastructure	12 s	12 s
Alias	52 s	52 s
Type	*N/A*	20 s
Symbolic	300 s	150 s

Table 10.2 Times Spent on Each Step of the Analysis

Chapter 11

Conclusion

In this project, we have managed to successfully develop a method for tracking runtime types for the Python scanner in Manta Flow. This method is fully integrated into the scanner and is slated to become a part of the Manta Flow production deployment in the near future.

We have provided a way to configure the functions that we cannot analyze directly through their source code in a way that allows for incremental improvement. We have implemented this configuration for the standard library plugin. With time, we hope to add this configuration for other libraries as well, which we believe will further improve the results of type analysis.

We were then able to use the results of this analysis to construct a more precise way of determining the possible targets of invocations. This also necessitated creating a way to parse the class hierarchy of an application. This improvement was integrated in a way that was seamlessly usable by the rest of the scanner. It is as of now the only use of type analysis, but we tried to design the system in such a way that any future uses can be easily accommodated.

This was not an independent work, but an extension of a complex preexisting body of software. As such, numerous other modifications, additions, and updates were necessary to implement our solution. We believe that as a whole, these changes noticeably improved the precision of the scanner and possibly also helped its performance, as we were able to empirically demonstrate. They also allowed us to improve the quality of the codebase, removing some vestiges of features that were either dependent on or made as a result of the imprecision in this area, and yet more updates in this vein are—in our estimation at least—still possible. The addition and integration of type analysis proved challenging enough that we unfortunately weren't able to make headway in adding support for callbacks and function pointers. This is another area of possible future improvement.

The nature of software is ever-changing, and Manta Flow with its Python scanner is being continually worked on and improved, so we have no doubt this is not the final iteration of type analysis and more updates will be made in the future. Nevertheless, we feel we have created a solid foundation that will allow the product to thrive and better serve the needs of its customers.

Bibliography

[1] Peter Naur and Brian Randell, editors. *Report on a conference sponsored by the NATO Science Committee.* 1968. URL: `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF`.

[2] Michal Jurčo. "Data Lineage Analysis Service for Embedded Code". Master's thesis. Chales University, Faculty of Mathematics, Physics, Department of Distributed, and Dependable Systems, 2023.

[3] Josef Kumstýř. "Precise and Efficient Incremental Update of Data Lineage Graph". Master's thesis. Chales University, Faculty of Mathematics, Physics, Department of Distributed, and Dependable Systems, 2022.

[4] Roman Firment. "Monitoring Support for Manta Flow Agent in Cloud-Based Architecture". Master's thesis. Chales University, Faculty of Mathematics, Physics, Department of Distributed, and Dependable Systems, 2022.

[5] *The Python Language Reference.* URL: `https://docs.python.org/3/reference/`.

[6] Michele Simionato. *The Python 2.3 Method Resolution Order.* URL: `https://www.python.org/download/releases/2.3/mro/`.

[7] Roger Hindley et al. "The principal type-scheme of an object in combinatory logic". In: *Transactions of the American Mathematical Society* 146 (1969), pages 29–60.

[8] John Aycock. "Aggressive Type Inference". In: *Language* 1050 (2000), page 18.

[9] *Pyan.* URL: `https://github.com/davidfraser/pyan`.

[10] "MaxSMT-Based type inference for Python 3". eng. In: *CAV (2).* Volume 10982. Springer International Publishing, 2018, pages 12–19. ISBN: 9783319961415.

[11] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. Edited by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 337–340. ISBN: 978-3-540-78800-3.

[12] C. M. Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal K. Roy. "Exploring Type Inference Techniques of Dynamically Typed Languages". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2020, pages 70–80. DOI: `10.1109/SANER48275.2020.9054814`.

[13] Collin Winter and Tony Lownds. *PEP 3107 – Function Annotations*. 2006. URL: `https://peps.python.org/pep-3107/`.

[14] Ryan Gonzalez et al. *PEP 526 – Syntax for Variable Annotations*. 2016. URL: `https://peps.python.org/pep-526/`.

[15] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. *PEP 484 – Type Hints*. 2014. URL: `https://peps.python.org/pep-484`.

[16] Philippe Prados and Maggie Moss. *PEP 604 – Allow writing union types as X | Y*. 2019. URL: `https://peps.python.org/pep-0604/`.

[17] Ivan Levkivskyi, Jukka Lehtosalo, and Łukasz Langa. *PEP 544 – Protocols: Structural subtyping (static duck typing)*. 2017. URL: `https://peps.python.org/pep-0544/`.

List of Figures

Listings

Appendix A

Attachments

A.1 User Documentation

There are several prerequisites to being able to generate data flow analysis using the Python scanner:

- Java 17 needs to be installed on your computer.
- Manta Flow needs to be installed on your computer. Manta Flow is proprietary software only available to developers and licensed customers of Manta.

A.1.1 Building and Running the Scanner

Since the Python scanner is a part of Manta Flow, and our work consists of changes made within the scanner, no explicit build is necessary—the scanner is accessible using the standard installation process.

If a user has Manta Flow installed, they can use the Python scanner by creating a new Python connection. This connection defines where the analyzed application is located, as well as some other properties used during its execution. The created connection is then used for creating workflows that extract and analyze the application, the results of which can be visualized using the Manta Flow Viewer.

A.2 Attachment Structure

The attachment published alongside this text consists of three directories.

The `source-code` directory contains the source code of files created within this work. This represents only a small fraction of the Python scanner, which cannot be included in its entirety. This directory contains a hierarchical structure that mirrors the placement of these files in the scanner.

The `diffs` directory contains changelogs encapsulating the full extent of changes made in this work, including changes to components created by other parties. Since these changes were integrated gradually alongside other development, it is impossible to condense them into a single file without inadvertently incorporating other changes in the process. To ease readability, the files in this directory were named so they appear in chronological order and the names broadly describe the contents.

Finally, the `tex-source` directory contains the full LaTeX source used to create this text, including any images contained within.

Appendix B

Library Type Sources

This appendix is related to the analysis we mention in Section 7.1. For each library supported by the scanner, we list where to find type hints for its public API.

`builtin`

The standard library.

Type Hints Source

The `typeshed` package, available at `https://github.com/python/typeshed`.

Notes

Official stubs for the standard library. Maintained directly by the Python team, very comprehensive. Some challenges in parsing it are described in Section 7.1.

`boto3`

A library for interfacing with AWS services using Python.

Type Hints Source

The `botostubs` package, available at `https://github.com/boto/botostubs`. Alternatively, the `mypy_boto3_builder` package, available at `https://github.com/youtype/mypy_boto3_builder`.

Notes

The `botostub` project is the official source created by Amazon. It is described as “a work in progress” and “not yet intended for production.” However, the last change in the project was made in 2019, which leads us to believe it was abandoned. The unofficial `mypy_boto3_builder` repository looks to be actively maintained and includes scripts to generate type annotations for the library.

`delta`

A storage framework for building lakehouses.

Type Hints Source

The source code, available at `https://github.com/delta-io/delta`, is annotated.

Notes

Some objects used only for type hints are imported conditionally only during type checking. Hints of such objects are specified as string literals so that they are resolvable expressions even at runtime. As a simple example of this:

```
if TYPE_CHECKING:
    from py4j.java_gateway import JavaObject

class DeltaTable(object):
    def __init__(self, spark: SparkSession, jdt: "JavaObject"):
        ...
```

Also uses types from other libraries, such as `JavaObject` from `py4j` or `DataFrame` from `pyspark.sql`.

`pandas`

A ubiquitous library for data analysis and data manipulation. Other libraries in this space tend to support working with its objects.

Type Hints Source

The source code, available at `https://github.com/pandas-dev/pandas`, is annotated. There is also a dedicated `pandas-stubs` package, available at `https://github.com/pandas-dev/pandas-stubs`.

Notes

The source code annotations are probably the most accurate, but they are not complete and are mainly for internal use. The `pandas-stubs` package provides stubs only for the public API and is intended for public use. As such, they aren't necessarily consistent with the internal type hints. They also don't cover every possibility of use available, though they aim to accurately describe the recommended practices, and they do not fully support version 2.0 of the library.

`pyspark`

Python API for Apache Spark.

Type Hints Source

The source code, available at `https://github.com/apache/spark`, is annotated.

Notes

Only the public API tends to be annotated, the internal classes are without type hints. Also makes use of string literals in types.

`sqlalchemy`

A database access library and an ORM manager.

Type Hints Source

The `sqlalchemy2-stubs` package, which is available at `https://github.com/sqlalchemy/sqlalchemy2-stubs`.

Notes

Despite the name, this package is only compatible with the 1.4 releases of the library. The 2.0 version—available at `https://github.com/sqlalchemy/sqlalchemy`—is typed fully inline and explicitly incompatible with these stubs.

`awsglue`

Python API for the AWS Glue ETL service.

Type Hints Source

No type hints appear to be available.

`dbutils`

Not a “real” package. It is a collection of utilities that are implicitly available in Databricks notebooks under this namespace. We have to support some of them when analyzing Python code that originated from Databricks.

Type Hints Source

No type hints are available.

Appendix C

Type Hinting Features

In this appendix, we go over every accepted and finished PEP in the *Typing* topic[1] that was added after the initial standardization of type hints in PEP 484, briefly describing its contents and the expected additional difficulty in extending an imagined parser for type hints to support it. We do not examine any purely informational PEPs or currently open PEPs, the former because they by definition do not add new features to the language, and the latter because they may be changed or withdrawn at any time.

PEP 526 — Syntax for Variable Annotations

This PEP extends the syntax described for function headers in PEP 484 to also apply to variables, allowing constructs such as `x: int = foo()` and thus declaring a type hint on a local variable.

Our analysis of how to use type hints focused only on type hints of functions, ignoring variable annotations. We can imagine a simple extension that could account for them though. If a type hint is found in an assignment, the type flow based on that hint would be propagated to the left-hand side instead of (or in addition to) the flows on the right-hand side.

PEP 561 — Distributing and Packaging Type Information

This PEP is concerned with ways type information is communicated. It codifies the concept of stub files and stub libraries and introduces some standards that allow developers (or tools) to discover the type hints for a given project.

[1] `https://peps.python.org/topic/typing`

While this PEP is useful in making it easier to find sources of type information, it has no impact on the syntax or semantics of the hints themselves, so no additional support should be needed.

PEP 585 — Type Hinting Generics in Standard Collections

This PEP allows parameterized generics in the standard library to be declared using directly those types, instead of relying on a parallel hierarchy in the `typing` package. For example, instead of having to specify `typing.List[str]` when annotating a list of strings, we can now directly write `list[str]`. This expression would not have been allowed before, as the `list` class object did not support the subscript operator.

Assuming the type hint analyzer can already process generics, it would only need to alias the collections so that they are handled the same way as their `typing` equivalents.

PEP 544 — Protocols: Structural Subtyping

This PEP introduces support for structural subtyping into the language. A protocol is defined by including the `typing.Protocol` class in the list of superclasses. When analyzing whether an instance object matches a type hint that uses a protocol, we do not look at the ancestry of that object's type. Instead, we check all methods defined on the prototype and verify that our type has a method with a matching signature for each of them.

It is clear that protocols introduce a completely new paradigm to type checking and as such would most likely require extensive support from the analyzer. We have not tried to describe the full extent of this support in this work. Luckily, protocols are not transitive (a subclass of a protocol is not itself a protocol unless it also includes `typing.Protocol` directly as a superclass), so there should at least be a somewhat straightforward way to ignore these type hints.

PEP 563 — Postponed Evaluation of Annotations

This is an implementation PEP that changes the way the annotations are stored and resolved to address issues such as forward references and the cost of evaluating the expressions. It should have no bearing on static analysis of type hints, other than allowing forward references, which is trivial and implicitly assumed in static analysis.

PEP 586 — Literal Types

This PEP adds the ability to specify that the annotated variable can only take on a specific value (or one of several specific values). For example, if a comparator function returns an `int` but we know that it can only return either -1, 0, or 1, we can make the annotation of this function more precise by specifying its return type as `Literal[-1, 0, 1]`.

The `Literal` class can only be parameterized by literal expressions. It is legal to write `Literal["foo", 2, False]`, but not `Literal[1+1]` or `Literal["foo".capitalize()]`. As such, adding support for this class should be relatively simple, as the types of literals are always readily apparent.

PEP 589 — TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys

This PEP creates a new type named `TypedDict` that allows a user to specify dictionaries with a known fixed set of keys. Imagine there is a dictionary object you know should always contain two keys: a `"foo"` key with an integer value and a `"bar"` key with a string value. Using only standard type hints, the best annotation we can provide is `dict[str, str|int]`. Such an annotation is unable to determine whether the dictionary has the correct keys, or whether each key has a value of a correct type.

The `TypedDict` type provides a way to specify those things explicitly. We can declare a class that inherits from `TypedDict` and specifies the names and value types of each key:

```python
class MyDict(TypedDict):
    foo: int
    bar: str

x: MyDict = {"foo": 2, "bar": "baz"} # correct
x = MyDict(foo=2, bar="baz") # alternative declaration
x: MyDict = {"foo": "baz", "bar": 2} # incorrect
x: MyDict = {"foo": 2, "xxx": "baz"} # incorrect
```

These classes can serve as superclasses for other such definitions, and the value types can be any valid annotation, including another `TypedDict` or even recursively the class it is declared on.

The simplest form of support for these hints would be to detect them and replace them with `dict[string, Any]`. A more full-featured solution would include properly parsing the class definitions and at minimum using a union of all the types that appear in it. Actually associating the types with individual keys would require even bigger changes, as type analysis is not equipped to differentiate between different keys of the same type.

PEP 591 — Adding a Final Qualifier to Typing

This PEP introduces the `@final` decorator and an associated `Final` annotation to the `typing` module. When a function is decorated with `@final`, it is a sign that the function should not be overloaded in any subclasses or overridden within its scope. Applied to a class it signifies that the class should not be subclassed. The `Final` annotation is used on variables or attributes to denote that they should not be reassigned. It takes a type argument noting the underlying type, e.g. `id: Final[int] = 1`. It can also be used without an explicit type, e.g. `id: Final = 1`.

We would not need to add any support for the `@final` decorator. It is used to enforce the correctness of code, which our scanner is not concerned with. For the `Final` annotation, we would merely have to extend the parser to unwrap it to get the underlying type hint, or to ignore it completely if none is provided.

PEP 593 — Flexible Function and Variable Annotations

This PEP introduces a mechanism that allows us to assign arbitrary annotations that can be used at runtime while still providing type hints for static analysis. It does so by creating the `Annotated` type. This type takes two parameters (i.e. `Annotated[T, x]`). The first is the type hint applied to the annotated function or variable, the second is an arbitrary expression. Static analyzers can restrict themselves to only check the former and still get the full benefits of type hinting, while at runtime we can create much richer metadata within the annotations.

In line with the semantics of this feature, the only support necessary here is to be able to replace the annotation with its first parameter.

PEP 604 — Allow Writing Union Types as `X | Y`

This is a syntax extension. Instead of having to specify a union of two types as `Union[X, Y]`, this PEP allows us to write this union using the form `X | Y`. Since this change only affects the syntax without any new semantics, the only necessity would be to support parsing both forms equivalently.

PEP 612 — Parameter Specification Variables

This PEP provides an extension of structural subtyping. Let us imagine we have a decorator that takes in a function and returns a new function with the same parameters but a different return type. The function parameter and return type can be annotated using the `Callable` prototype, however, prior to this feature, there was no way to specify that they both accept the same number and types of parameters. This PEP introduces `ParamSpec` variables, which can be used for this purpose.

As the prerequisite for supporting this feature is support for structural subtyping, we have not made a full evaluation of this feature in this work.

PEP 613 — Explicit Type Aliases

This PEP adds a `TypeAlias` annotation that explicitly marks an assignment as declaring a type alias. When making an assignment such as `X = int`, it is clear that `X` is a type alias and later annotations such as `foo: X = 1` are valid. However, when the assignment becomes `X = "int"`, possibly because the aliased class is not yet fully defined and this is a forward reference, a type checker may confuse this with a simple value assignment and then later raise errors when the alias is used in an annotation. Using the annotation, `X: TypeAlias = "int"` lets the type checker explicitly know to treat this assignment as a type alias.

Assuming the type hints parser supports type aliases, and assuming it supports string literal type hints but in a way that does not let us combine the two, the `TypeAlias` hint could be used to bridge that gap. Otherwise, it does not serve much purpose for our use case.

PEP 646 — Variadic Generics

When declaring a generic class, it might have a variable number of type parameters. This PEP allows for a new syntax, `Generic[*T]`, to denote that a specification of the generic type may contain any number of parameters of type `T`. The types captured by this variadic type parameter can then be annotated as `Tuple[*T]` within the class.

How this feature is supported would depend entirely on how generics would be handled. It might necessitate adding new types of flows and propagation rules, or it might be solved by simple replacements during parsing.

PEP 647 — User-Defined Type Guards

We might have an object that is annotated with a general type, but based on a runtime check determine it is actually of a more specific type. The canonical example of this is the `isinstance()` function. If we are given a hint of `x: object`, we have to treat `x` as the most generic type of all. However, if we later encounter a statement such as `if isinstance(x, str):`, we know that within the block under that check we can treat `x` as though it were a string.

While type checkers tend to support the `isinstance()` function for these situations, there may be more complex, user-defined guards that would be far more difficult to evaluate. To allow us to explicitly annotate such guards, this PEP adds a new `TypeGuard` annotation. When applied to the return value of a function, this annotation lets us know that not only does the function return a boolean value, but that if it returns `True`, the supplied argument is of a specific type. For example, a function that takes in a `list[object]` parameter and checks whether all values in the list are strings would be annotated as returning `TypeGuard[list[str]]` If such a function were to be later used as a condition of an `if` statement, the type checker would know to treat the passed argument as `list[str]` within the block.

The benefits of supporting this feature beyond replacing the annotation with `bool` are dubious. The scanner is built on the idea that it is not context-sensitive, so we would have to build mechanisms that allow us to make an exception in this case. We would also have to somehow circumvent the limitation of the worklist algorithm, which does not allow us to remove flows from the flow set. As such, the costs would seem to outweigh the benefits for anything more than the aforementioned simple replacement.

PEP 649 — Deferred Evaluation of Annotations Using Descriptors

Similar to PEP 563, this is another implementation-focused PEP that changes how annotations are stored and resolved at runtime. It does not concern us.

PEP 655 — Marking Individual `TypedDict` Items as Required or Potentially-Missing

This is an extension of the `TypedDict` feature added in PEP 589. It defines two new annotations, `Required` and `NotRequired`, to specify that a given key is either explicitly required in a partial definition or not required in an otherwise complete definition.

Depending on how `TypeDict` is supported, these annotations may not require any support, as they can only appear within the items of a `TypedDict`. Even if an analysis of these items was implemented, we always have to consider all keys as possibly present, and we don't care about the correctness of the analyzed code, so we could safely ignore these annotations.

PEP 673 — Self Type

This PEP introduces a simple way to annotate methods that return an instance of their class. By specifying the return type as `Self`, we can determine the method returns an object of the same type as that of the caller (most commonly `self`, the caller itself, hence the name). There is a slight difference between this annotation and simply writing the name of the enclosing class, which is caused by subclassing.

When we have a class `A` with a method `foo()` that just sets some property on the caller and then returns it, if we annotated it as returning `A`, a type checker would consider that as literally returning instances of `A`. This could become incorrect if we introduced a new class `B` that inherits from `A`. After calling `foo()` on an instance of `B`, we would no longer be able to call any methods that don't appear in `A` without causing an error. Annotating `foo()` as returning `Self` solves this issue.

This seems like a problem initially, however as we noted in our initial analysis of type hint support, we would already have to consider each return type as an upper bound rather than an explicit type, so we would be able to simply replace each instance of `Self` with the type of the class it appears in.

PEP 675 — Arbitrary Literal String Type

This PEP adds a `LiteralString` type hint that specifies that a function parameter must be a string literal without limiting the value of that literal. This feature has some use cases, such as helping prevent SQL injection by disallowing concatenated strings as database queries, but in our analysis, we can treat such hints exactly the same as `str`.

PEP 681 — Data Class Transforms

This PEP describes a new `@dataclass_transform` decorator that, when applied to a decorator or a metaclass, informs the type checker that decorated classes are imbued with "dataclass-like" functionality at runtime. This includes creating an implicit `__init__()` implementation and value-based equality methods.

Supporting this decorator could be interesting for symbolic analysis, especially for properly handling the implicit initializer, but does not seem particularly notable for type analysis.

PEP 692 — Using `TypedDict` for More Precise `**kwargs` Typing

When we annotate the `**kwargs` parameter of a function with type `T`, it is interpreted as all values passed into that parameter as being of type `T`. The actual type of the `**kwargs` parameter is then `dict[str, T]`. To allow more granular control over values passed into this parameter, this PEP adds the ability to specify a `TypedDict` definition as the type hint for `**kwargs` instead.

Assuming support for `TypedDict`, this feature should require no major updates. In fact, it is the "standard" behavior that needs explicit support to convert into the real type. Enabling this feature would just mean turning that conversion off if the annotation represents a `TypedDict`.

PEP 695 — Type Parameter Syntax

This PEP adds new syntactic elements that simplify the specification of generic functions and classes, especially the definition and application of type variables and type aliases. There are several changes introduced in this PEP, but none of them changes the semantics of type annotations, so respective additions would only need to be made to the code parsing these annotations.

PEP 698 — Override Decorator for Static Typing

This PEP introduces a new `@override` decorator that explicitly marks a function as overriding a function of the same name in a base class. This feature is primarily useful to prevent bugs caused by changing the API of the base function without reflecting the change in its inheritors, but it has no impact on the runtime, data lineage, or the signature of the decorated function, so we can safely ignore it.

PEP 702 — Marking Deprecations Using the Type System

Similar to the previous PEP, this one introduces the `@warnings.deprecated()` decorator that marks a function as deprecated. This decorator can be used by static checkers to raise a warning when a decorated function is used, and it can emit a warning during runtime, but otherwise, it has no bearing on the actual execution and can be safely ignored.