Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelor Thesis in Computer Science

Development of a Data Provenance Analysis Tool for Python Bytecode

Nadejda Ismailova

1 Mai 2015

Reviewer

Prof. Dr. Torsten Grust
Database Systems Research Group
University of Tübingen

Supervisor

Tobias Müller
Database Systems Research Group
University of Tübingen

Ismailova Nadejda
Development of a Data Provenance Analysis Tool for Python Bytecode
Bachelor Thesis in Computer Science
Eberhard Karls Universität Tübingen
Period: 01.01.2015 - 01.05.2015

Abstract

Data Provenance describes the origins of data. In this thesis we propose an approach of computing data provenance when data of interest is created by a Python program and present a tool implementing this approach. In implementing the tool, we adopt the following idea. In the first step, the program is instrumented to log data. In the second step, the code is analyzed with respect to the logged data and provenance information is computed. A novel aspect of this work is instrumentation and analysis of *the bytecode* of the given Python program with the aim of computing data provenance.

Contents

1 Introduction

Data provenance provides historical and contextual information about the source of the data of interest and the way it was produced. This general concept was studied in and applied to many different contexts and use cases, such as business applications, the Web, distributed systems, databases and many others. In databases, *why*, *where* [CCT09] and *dependence* [CAA07] provenance notions have been proposed, which explain the output data of a query in terms of the input data. We will focus on these provenance notions.

In this thesis, we develop a tool for computing data provenance in a Python program. The tool will be a part of a workflow for visualizing provenance of a given SQL query. Figure 1.1 illustrates the workflow(the right branch).

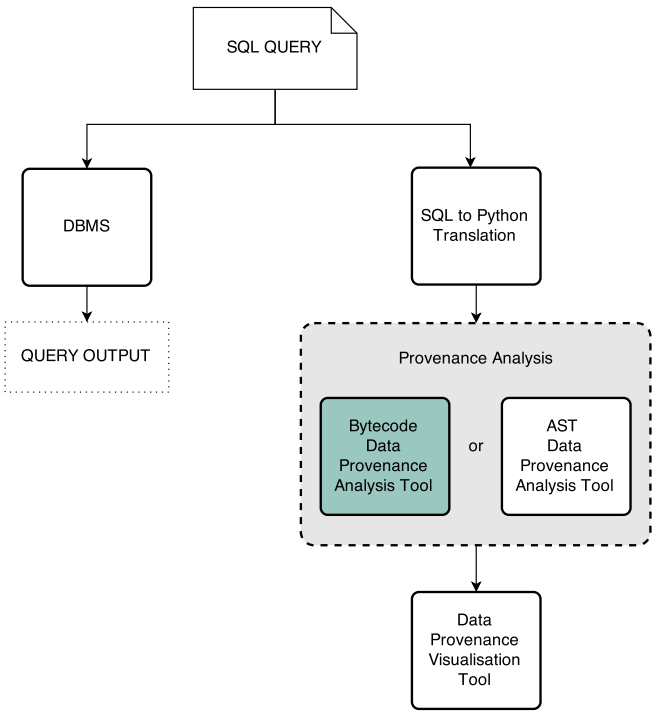


Figure 1.1: Project context

In the workflow, the SQL query of interest is translated to a Python program. The SQL Compiler doing that is still in development.

As next, data provenance is computed through analysis of the generated

Python program. Currently, this task is done by the AST Data Provenance Analysis Tool, which works on the Abstract Syntax Tree representation of the program. In this work we develop an alternative Bytecode Data Provenance Analysis Tool, which will derive provenance by processing *the bytecode* of the given program.

Finally, analysis results can be visualized in a web-based Data Provenance Visualisation Tool proposed by [Bet14].

The tool proposed here is computing provenance in two steps. Instrumentation of the bytecode, so that additional information is logged at run time, is the the first step. Symbolic execution, which derives dependencies between input and output data, is the second step.

The document structure is as follows.

In Chapters 2 - 4 we provide required background information and the basic techniques needed for the implementation. Here, Chapter 2 introduces data provenance notions and a possible interpretation of it in terms of Python programs. Chapter 3 presents the general algorithm for computing provenance in program context. Finally, Chapter 4 gives an overview of the Python compilation process and shows what bytecode instrumentation is.

Chapter 5 is devoted to the implementation.

At last, we give our conclusions and propose some issues for the future work in Chapter 6.

The implementation of the tool is leant on the existing AST implementation and have the same input and output interfaces. For this reason, several modules (e.g. for database communication) of the AST-Implementation were adapted for use in the current implementation.

Screenshots of the visualisation [Bet14] are used throughout the work.

2 Data Provenance

Chapter Outline

The structure of this chapter is as follows.

In Sections 2.1 and 2.2 we first introduce the notion of data provenance. We then provide an overview of the different kinds of data provenance and describe them with the help of an example. Finally, we propose how the data provenance notions mentioned can be understood in terms of a given Python program and transfer the data provenance concept to dependencies between argument and return values.

In Section 2.3 we show the different perspectives of the dependencies between program arguments and program's return value(s), which are eventually the output of the analysis tool.

2.1 Notion of Data Provenance

Data provenance (also called *lineage*) records the source (where does the piece of data come from), derivation (the process by which the piece of data was produced), or other historical and contextual information about the data. We can find many forms of it in our daily life. For instance, a simple and familiar example is the time-date and ownership information of the files in the file systems.

Such information can be useful in numberless applications and contexts. Accordingly, the topic has been the focus of many research projects and prototype systems in many different areas. However in this work we focus on the notions of provenance as described in the context of databases [BKWC01], [CCT09], [CWW00] .

2.1.1 Why, How and Where

In the databases context provenance has been studied inter alia from three perspectives, that, intuitively, describe following relationships between the input and the output data sets [1]:

- *Where Provenance:* where does a piece of output data set come from the input data set? In other words, where-provenance points to the location in the input, where the output values were copied from.
- *Why Provenance:* why is a piece of data in the output data set? Thus, why-provenance refers to input values which influence the occurrence of data of interest in the output.
- *How Provenance:* how was a piece of output data set produced in detail?

Our interpretation of where provenance is extended to include not only locations of copied values, but all input parts participating in computation of the relevant output value.

Whereas we discuss why and where provenance at some points in the next chapters, how provenance is not considered in the rest of the thesis.

[1] For more detailed and/or formal description see [BKWC01], [CCT09].

2.1.2 Example

We will illustrate the idea of why and where provenance through an example from [CCT09]. Consider the following query:

Listing 2.1: Example query

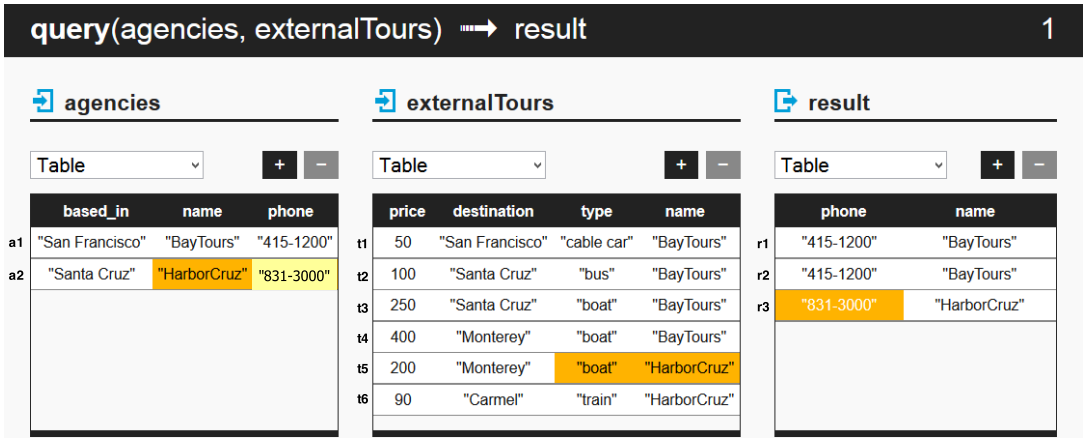
```
SELECT a.name, a.phone
FROM Agencies a, ExternalTours e
WHERE a.name = e.name AND e.type="boat"
```

Suppose that the query is executed on tables **agencies** and **externalTours** as in Figure 2.1, where the labels **a1**, … , **r3** are used to identify the records. In the figure why provenance is highlighted orange and where provenance is highlighted yellow.

The results of the query are shown in the table **result**. As we can see, the query returns names and phone numbers of all agencies offering boat tours.

If we did not expect the "HarborCruz" to be an agency offering boat tours, we could ask ourselves why it is in the output. To get the answer, we look at the why provenance of this record. In Figure 2.1 it is highlighted orange (in this case the field has the same why provenance as the record containing it). We see that the field **r3.phone** is included in the result because **a2.name**, **t5.name** and **t5.type** values, which are relevant for the join and filtering conditions of the query, qualify that record to be returned in the output.

Now we might like to know, where the field **r3.phone** is coming from. For instance, when the number is not correct, we would like to fix it in the source table. In such cases where provenance provides the information we need. Figure 2.1 shows that the field in the result is copied from **a2.phone** field in **agencies**.



	based_in	name	phone
a1	"San Francisco"	"BayTours"	"415-1200"
a2	"Santa Cruz"	"HarborCruz"	"831-3000"

	price	destination	type	name
t1	50	"San Francisco"	"cable car"	"BayTours"
t2	100	"Santa Cruz"	"bus"	"BayTours"
t3	250	"Santa Cruz"	"boat"	"BayTours"
t4	400	"Monterey"	"boat"	"BayTours"
t5	200	"Monterey"	"boat"	"HarborCruz"
t6	90	"Carmel"	"train"	"HarborCruz"

	phone	name
r1	"415-1200"	"BayTours"
r2	"415-1200"	"BayTours"
r3	"831-3000"	"HarborCruz"

Figure 2.1: Why and where provenance

Note that the result contains duplicate entries. A reasonable question at this point might be: why does the "BayTours" entry occur twice in the output?

For those who wonder, our query semantics is not eliminating duplicates from the output. We use this fact to show that with the help of provenance (why provenance in this case) we can explain why there are duplicates in the result in a very simple way. A name of an agency and a tour type do not identify a single row in the join table **externalTours** since there are boat tours to different destinations, that is why we get the "Bay Tours" entry twice. And that is exactly what we see in the figure shown below. In Figure 2.2 it is easy to see that the existence of each of duplicate rows in the output is justified by a different row in the input **externalTours** table.

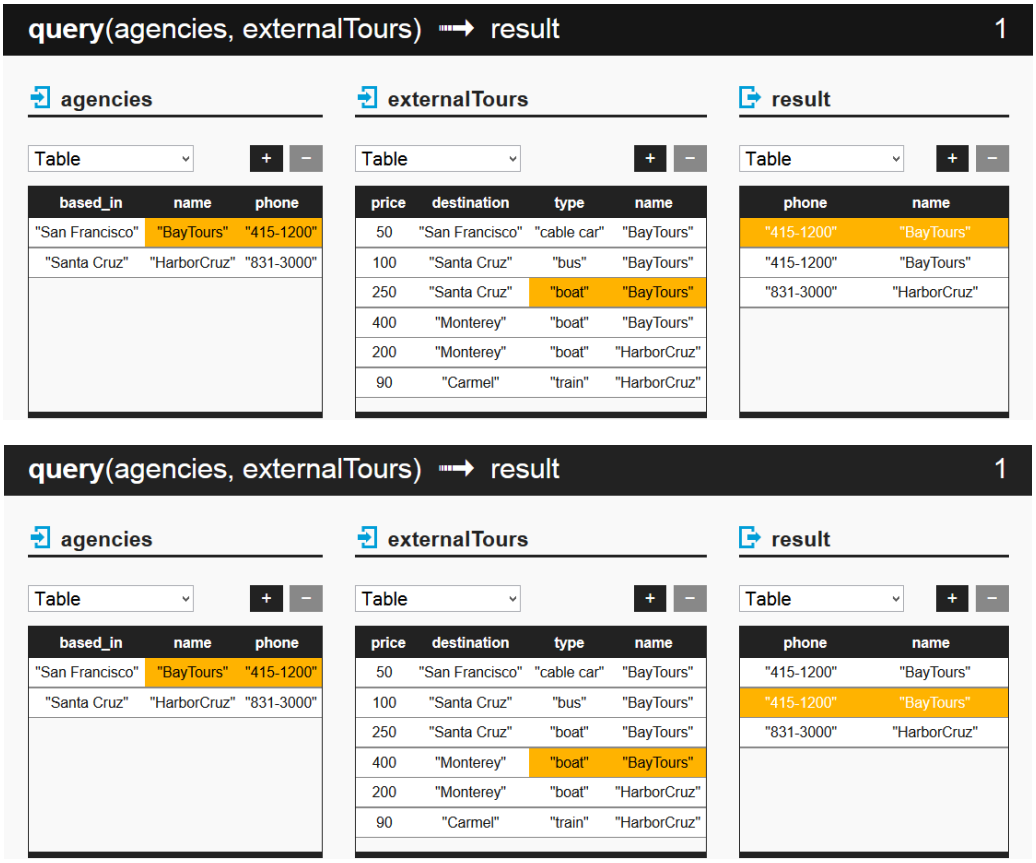


Figure 2.2: Duplicate entries have different provenance values

The presented example is simple, but if we consider that the most real-world databases have hundreds of columns and millions of rows, than an explanation to the relationships between each part of input and output seems correspondingly useful and important.

2.2 Program Slicing and Dependency Provenance

A more general perspective on data provenance was also introduced [CAA07]. To understand it, we start with a few words about program slicing.

2.2.1 Program Slicing

Program slicing [Wei81] is a form of program analysis that identifies program parts which contributed to program results, primarily for debugging purposes. As in other program analysis techniques dependence is a key concept in slicing [ABHR99], describing how variables or control flow points influence other program parts.

- A *slice* to some variable a consists of all program statements that somehow affect the value of a, or, in other words, on which variable a *depends*.

Naturally, we want slices to be as small and thus easy to understand as possible, so the program itself as a trivial slice is not of interest.

In cases where program has conditional expressions, we differentiate between *static* and *dynamic* slices. In case of a static slice we don't know which branch was taken at run time and so consider instructions and dependencies in both branches. In contrast, dynamic slices only contain statements from the actually executed branch, statements and dependencies in branches that were not taken are ignored.

As an example, suppose a program as in Listing 2.2:

Listing 2.2: Function takes a list of numbers as input and returns a list where numbers are reordered in a way that even numbers come first in the list.

```
def sortOnEvenNumbers(numbers):

    numElements = len(numbers)
    # index for even numbers
    evenIndex = -1
    # index for odd numbers
    oddIndex = numElements
    # result list, initialized with zeros
    sortedList = [0] * numElements

    # index variable for iteration over the input list
    i = 0
    while i < numElements:
        # read next list's element
        n = numbers[i]
        # got an even number
        if n % 2 == 0:
            # fill list from the left
            evenIndex += 1
```

```
20                sortedList[evenIndex] = n
21            # got an odd number
22            else:
23                # fill list from the right
24                oddIndex -= 1
25                sortedList[oddIndex] = n
26            i = i+1
27
28        print sortedList
29
30 # input
31 numbersList = [1,2,3,4,5,6,7,8,9,10]
32 # function call:
33 sortOnEvenNumbers(numbersList)
34
35 # output:
36 # [2, 4, 6, 8, 10, 9, 7, 5, 3, 1]
```

The program just moves even numbers in a given list to be at the beginning of the list and odd numbers to come afterwards.

Following Listings give examples of static and dynamic slices to different variables of the function in Listing 2.2. Statements not included in a slice are shown grayed.

Listing 2.3: Slice with respect to index variable *i*

```
1 numElements = len(numbers)
2 evenIndex = -1
3 oddIndex = numElements
4 sortedList = [0] * numElements
5
6 i = 0
7 while i < numElements:
8     n = numbers[i]
9     if n % 2 == 0:
10        evenIndex += 1
11        sortedList[evenIndex] = n
12    else:
13        oddIndex -= 1
14        sortedList[oddIndex] = n
15    i = i+1
16
17 print sortedList
```

Listing 2.4: Static slice with respect to *sortedList*

```
1 numElements = len(numbers)
2 evenIndex = -1
3 oddIndex = numElements
4 sortedList = [0] * numElements
```

```
5
6 i = 0
7 while i < numElements:
8     n = numbers[i]
9     if n % 2 == 0:
10        evenIndex += 1
11        sortedList[evenIndex] = n
12    else:
13        oddIndex -= 1
14        sortedList[oddIndex] = n
15    i = i+1
16
17 print sortedList
```

Listing 2.5: Dynamic slice with respect to *sortedList* with i = 2, i.e. n = 3

```
1 numElements = len(numbers)
2 evenIndex = -1
3 oddIndex = numElements
4 sortedList = [0] * numElements
5
6 i = 0
7 while i < numElements:
8     n = numbers[i]
9     if n % 2 == 0:
10        evenIndex += 1
11        sortedList[evenIndex] = n
12    else:
13        oddIndex -= 1
14        sortedList[oddIndex] = n
15    i = i+1
16
17 print sortedList
```

Listing 2.6: Dynamic slice with respect to *sortedList* with i = 5, i.e. n = 6

```
1 numElements = len(numbers)
2 evenIndex = -1
3 oddIndex = numElements
4 sortedList = [0] * numElements
5
6 i = 0
7 while i < numElements:
8     n = numbers[i]
9     if n % 2 == 0:
10        evenIndex += 1
11        sortedList[evenIndex] = n
12    else:
13        oddIndex -= 1
14        sortedList[oddIndex] = n
15    i = i+1
16
17 print sortedList
```

Static slice to `sortedList` omits only the `print sortedList` statement at the end of the program, since it is the only statement not influencing `sortedList`. But dynamic slices are a bit more simplified, because according to condition evaluation in the `if` clause only one branch affects the value of `sortedList` and another branch is omitted.

2.2.2 Dependency Provenance

As we know by now, provenance explains the results of a query in terms of the input data. Intuitively, we say that the input has contributed to or has influenced or is relevant to the output. Cheney et al. generalize that intuition by defining dependence between output and input if a change on the input may result in a change to the output [CAA07].

Here we see an outstanding similarity between program slicing and data provenance. In program slicing as well as in data provenance we are looking for parts contributing to the output, with the difference that provenance identify parts in the input database and slicing in the program. The analogy is more detailed discussed in [Che07].

Similar to dependence concept in slicing dependence provenance is defined, formal details to which were proposed by [CAA07]. Intuitively,

- *Dependency provenance* is information relating each part of the output of a query to a set of parts of the input on which the output part depends [Che07].

As an example, consider an output record r with a field A and an input record s containing field B. Due to the definition above, r.A depends on s.B if changing s.B in some way either removes record r from the output completely or changes the value of r.A. The dependency provenance of r.A is then the set of all such input fields s.* on which r.A depends.

The following notation is used throughout the thesis. For some value A which depends on B we write the dependency as a tuple (A,B). For some value A depending on several values B,C,D, ... a dependency provenance set is given as $\{B, C, D, \ldots\}$ to list all the dependencies.

Additionally, whenever the phrase “dependency-correctness” is used in the rest of this thesis, we intend that the provenance set computed for some output value contains all parts of the input which may lead to a change of the output value (compare [CAA07]).

2.2.3 Example

For convenience we are going back to our example from Section 2.1.2.
For ease of presentation in all the following figures same (orange) highlighting is used for all dependency provenance items.

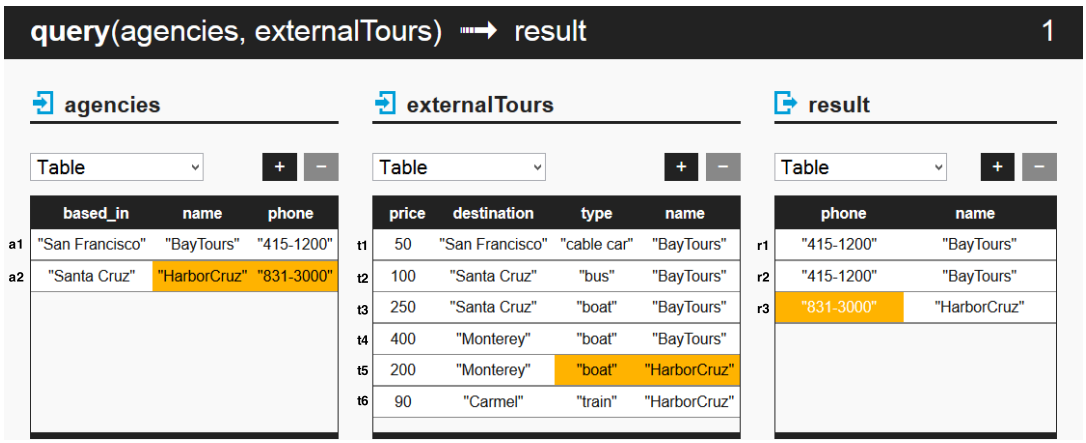


Figure 2.3: Dependency provenance of **r3.phone**

The dependency provenance of the result field **r3.phone** is the set { **a2.name**, **t5.name**, **t5.type**, **a2.phone** }, since changing these values can affect the result. In more detail: **r3.phone** is copied from **a2.phone**, so changing **a2.phone** would change **r3.phone**. Fields **a2.name**, **t5.name** are participating in the join and **t5.type** in the filtering condition, that is changing these values can exclude **r3** from the output.
Dually, **r3.phone** does not depend on all the other fields, what means, that after any modifications on these fields the **r3.phone** value of the "HarborCruz" entry would be still the same and would still be part of the output.

2.2.4 Limits of the model

Cheney et al.[CAA07] showed that obtaining "minimal" dependence information is undecidable. So our goal is to approximate the dependency-correct provenance minimum for each of the output items.

2.3 Provenance in Python Programs

Although the where and why provenance can be interpreted and represented in terms of Python programs, the concept proposed here is based on the idea of dependency provenance. We yet tried to keep the architecture adaptable for why and where differentiation in the future.

Speaking about provenance in a *program*, dependencies within a single Python *module* are meant, thus inter-module dependencies are not analyzed.

2.3.1 Dependencies

All the provenance definitions from above stay the same in terms of Python programs, with the difference that we no longer consider fields or records or relations. Instead we concentrate on variables (their values more precisely), what in effect is more general, since the value a variable points to can represent a single field or a single record(row) or the whole relation(table). Note that dependence sets are still derived for data, i.e. values variables are pointing to. So if at some point in the program a variable is re-bound and thereafter points to another value, then a different dependence set is associated with the variable, namely, the dependence set of the value currently bound to the variable name.

If a variable is pointing to some kind of collection (tuple, list, dictionary), then we derive dependencies for each of the elements in the collection. In the remainder of the work an element of a collection is identified by its "path", see Listing 2.7.

Listing 2.7: Identifying collection items through their paths

```
# Suppose a list of dictionaries:
list = [{"key1": 1}, {"key2": 2}, {"key3": 3}, {"key4": 4}]
# In Python we read the value 2 as follows:
theTwo = list[1]["key2"]
# The path to the value 2 is:  ['list', '1', 'key2']
# The same for the value 4: ['list', '3', 'key4']
```

Since provenance explains output in terms of input, dependence set of the return value (and dependence sets of all collection elements if a collection is returned) is what we are aiming for.

Consider the following Python program:

Listing 2.8: Dependencies in a Python Program

```
def sumUpMultiples(numbers, factor):
    sum = 0
    i = 0
    numRows = len(numbers)
```

```
5     while i < numRows:
6         n = numbers[i]
7         # filtering condition:
8         # here the decision is made,
9         # whether an element of the input list
10        # will contribute to the output value
11        # <-> why provenance
12        if n % factor == 0:
13            # input list's element n is
14            # added to the return value sum
15            # <-> where provenance
16            sum += n
17        i = i+1
18
19    return sum
20
21 numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
22 print sumUpMultiples(numbers, 3)
23
24 # output:
25 # 18    <-> 3 + 6 + 9
```

The function takes a list of `numbers` and a `factor`. It returns the sum of all numbers in the list, which are multiples of `factor`.

With the given input the function computes the sum of all the multiples of three in the number sequence 1 - 10, that is $3 + 6 + 9 = 18$.

How is the `sum` computed? Well, we iterate over the numbers list, check whether a number is a multiple of the `factor` variable's value and add it to the `sum` variable if it is. So what we return is computed from a set of input list items and the `factor` decides which items these are. Hence, the `sum` depends on those list items, which contributed to its value, and on `factor`, which decides which items to involve into computation.

For our example input this means: the returned `18` depends on values `{3,6,9,3}`. Both `3` entries are required, because they are indeed two different entities: one is the input item from the `numbers` list and the second is the value of the `factor` variable. Thus, the dependence provenance set of the `sum` variable is `{['numbers', '2'], ['numbers', '5'], ['numbers', '8'], factor}`.

There are two significant points in the program influencing `sum` variable's provenance.

On one hand, the `if` clause in line `12` is filtering list items to multiples of `factor`. If one would ask 'why is `9` added to `sum` ?' the answer would be 'Because `9` is a multiple of `3`'. That is, for an `n` that satisfies the condition, values of both `n` and `factor` are representing the why provenance of `sum`.

On the other hand, the `sum += n` statement, which is actually computing the

output from the input. According to our interpretation of provenance, whatever `n` is pointing to in the moment of execution is part of the where provenance of `sum`.

In other words, an interesting and reasonable interpretation is to describe why provenance in a program by control dependencies and where provenance by data dependencies.

The program analyzed here only contains one function call. However, a program in general can generate output by modifying the original input data through several sequential calls to different functions or nested calls within one function. The handling of dependencies in this context is discussed in the next section.

2.3.2 AA, AR, RA, RR Dependencies

In the last section an interpretation of dependence provenance in a Python program was proposed, where a single function call computed the output. More complex programs involve several steps, i.e. multiple sequential function calls. Furthermore, since the tool is supposed to analyze translated SQL queries, it seems eligible to assume, that nested queries will occur, which probably would be translated to nested function calls.

In such cases we still want to trace the final program output back to the original program input. But, additionally, the intermediate results, i.e return values of all the called functions, should be kept and explained in terms of the input of the particular function too, as well as the data flow between single function calls [2].

In this context, following four perspectives of dependency relationship between input, i.e. function arguments, and output, i.e. function return value, are proposed:

- *RA* - **R**eturn value on **A**rguments dependence:
  (return value, argument)
  They describe how the return value of a function depends on the arguments of the same function. This is the base case we saw in Section 2.3.1.
- *AR* - **A**rguments on **R**eturn values dependence:
  (argument, return value)
  When an argument of a currently executed function somehow depends on a return value of some other previously called function, we call it an AR dependency.

[2] Talking about *functions* here and in the rest of the thesis, Python's *user defined functions* are meant, see "Callable types" in the Python Documentation [Fou13a].

- *AA* - **A**rguments on **A**rguments dependence:
  (argument, argument)
  This kind of dependencies form when a nested function call occurs and the arguments of the called function somehow depend on the arguments of the caller.
- *RR* - **R**eturn value on **R**eturn value dependence:
  (return value, return value)
  Again, this kind of dependencies form when a nested function call occurs. Here, the return value of the caller depends on the return value of the called function.

These dependencies are the output of the tool developed in this thesis.

Observe that only for RA dependencies both parts of the relationship, return value *and* argument, belong to the *same* namespace. In the remaining cases elements from at least two *different* namespaces are somehow interacting.

Here, it is important to know, that we describe the relationship always from the perspective of the current namespace, i.e. namespace currently being analyzed: (value in current namespace, value in a different namespace).

Note also that all four kinds of dependencies imply both why and where provenance, thus we have AA why and where provenance, AR why and where provenance, RA why and where provenance, RR why and where provenance.

In the remainder of the thesis, the (sub)set of all dependencies in a program is noted as $\{(v1, v2), (v2, v3), (v4, v5), \ldots\}$, where $(v1, v2)$ is one of dependencies defined above between values $v1$ and $v2$.

As mentioned, we have illustrated RA dependencies in the program in Section 2.3.1 already. Following examples make the idea of the remained AR, AA, RR dependencies clear.

Sequential calls

AR dependencies occur, when a return value of a function influence which arguments another function called later will get. Look at the Figure 2.4:

In the figure, three functions are called sequentially. The different namespaces are illustrated as grayed boxes.

The `firstPassThrough` function gets a list `[1, 2, 3, 4]` as input. Obviously, it just forwards the input to the output. The blue arrows illustrate the RA dependencies.

The next function call `filter()` takes the return value `res1` of the previous function as an argument. Thus, this argument depends on the return value of the other function. The red arrows represent these AR dependencies: the first list item of the input list of `filter()` depends on the first list item of

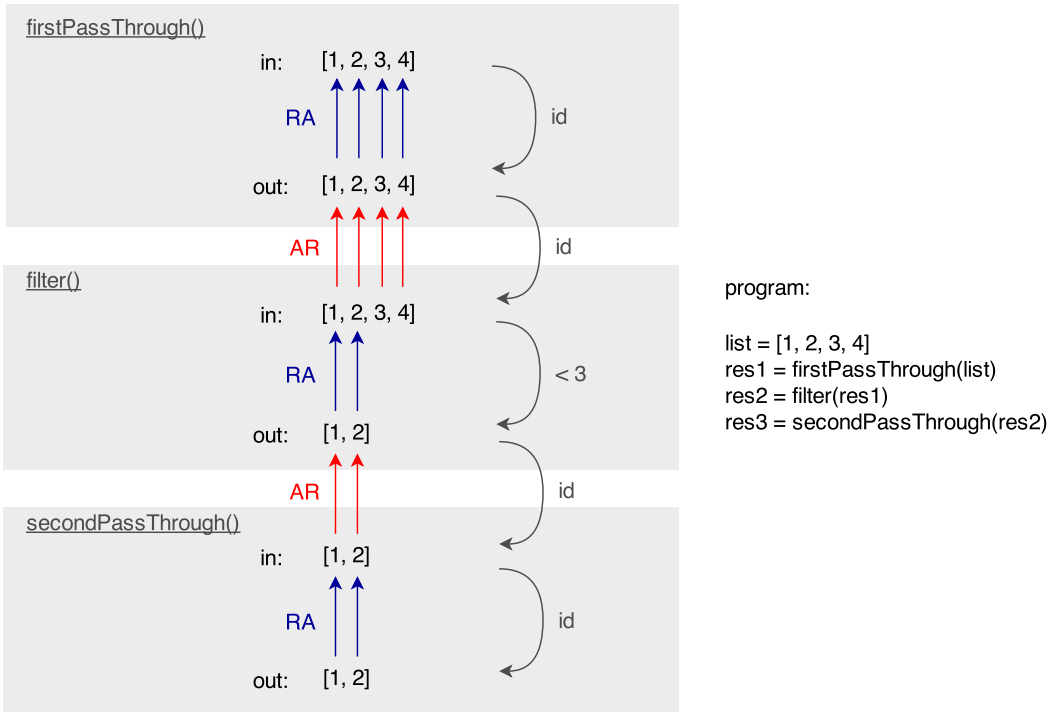


Figure 2.4: RA and AR dependencies

the output list of `firstPassThrough() - (1,1)`, the second list item of the input list of `filter()` depends on the second list item of the output list of `firstPassThrough() - (2,2)`, and so on: `(3,3)`, `(4,4)`.

The `filter()` function returns only items from the input list, which are `< 3`, i.e `[1, 2]`. This return value is then passed as argument to the `secondPassThrough()` function, that is, we have AR dependencies again. Input of the `secondPassThrough()` depends on the output of the `filter()`: `{(1,1),(2,2)}`.

The `secondPassThrough()` only returns its own input, again.

Now, when we look at the `1` at the very end, returned by `secondPassThrough()`, and follow the arrows representing the AR and RA dependencies, we come to the very first `1` defined in the `list`. By this means, we tracked all the transformations the output value `1` was processed by back to its origin.

Nested calls

AA and RR dependencies occur, when a function has recursive calls or calls to some other functions within its body. The Figure 2.5 below shows one possible scenario.

Two functions `passThrough()` and `filter()` are called sequentially. However, now `filter()` calls to another function `nestedFunctionCall()` inside

its body. Thereby, `nestedFunctionCall()` takes input of the `filter()` as own input, these are AA dependencies shown by green arrows. The output of the `nestedFunctionCall()` is returned by `filter()` as own output, which is shown by the orange RR dependencies arrows.

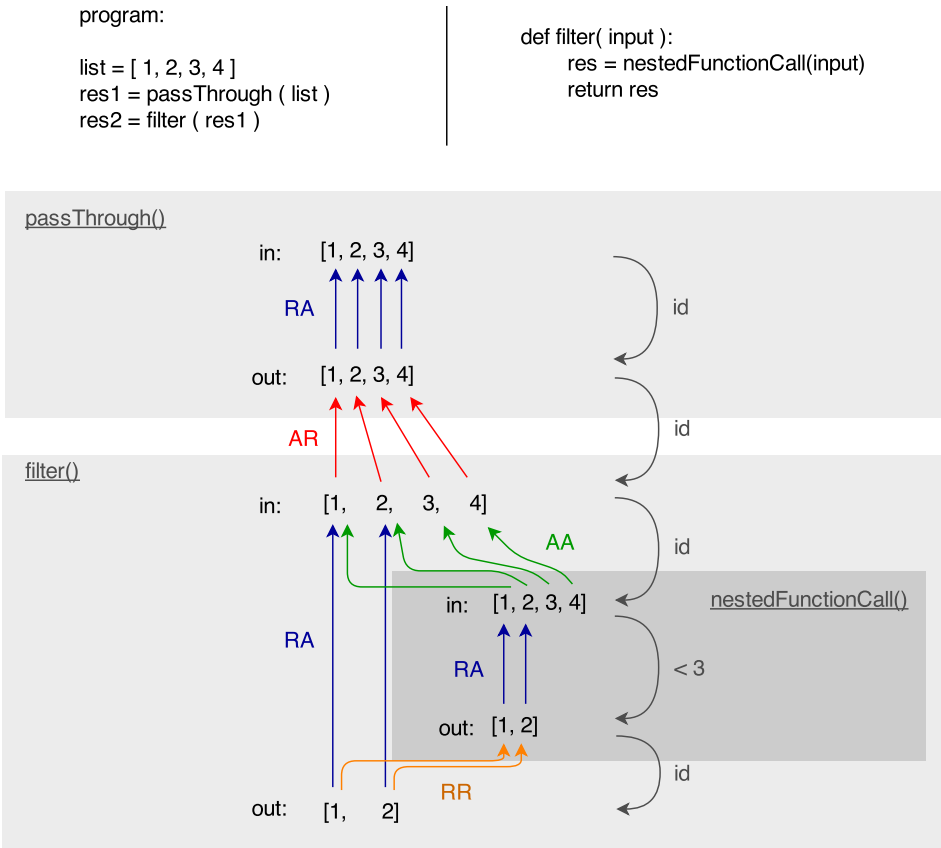


Figure 2.5: RA and AR dependencies

Again, at the end each of the `[1, 2]` output items can be tracked to its origin by following the dependency arrows.

3 The Algorithm

Chapter Outline

In this Chapter the algorithm how to derive dependence provenance for a given program is proposed.

The first section describes the general procedure and gives an overview of the components.

In the rest of the Chapter we take a closer look at the instrumentation and the dependency analysis steps.

3.1 General Approach

In this thesis, provenance is computed through *value-less* or *symbolic execution* of the input program. Here, *execution* means, that while evaluating the program, we follow the same control flow and access the same subscript elements, as it would be done at run time. *Value-less* means that we do not involve the actual *values* into computation, which were given by the input or defined by the program. Instead, *dependencies* are "assigned" to the variables and the data processing steps in the program are interpreted in terms of these dependencies. The output produced this way contains the provenance of the output value produced by the real execution.

Yet, before we can start that simulated "execution", we need to know the following:

- control flow decisions, i.e. whether an if-block or a loop body was executed
- collection access, i.e. subscripts used to read or write collection items, for example, value of the index variable `i` when accessing `someArray[i]`

To get this information, we have to actually run the program with the input data of interest and somehow remember relevant data.

Therefore, the whole procedure is split up into two steps, see Figure 3.1.

We get the program to analyze as input.

In the first step, illustrated in the green area of the diagram, we *log run time data*. This is done as follows. First, we *instrument* the input program, i.e. extend it with additional functionality. Then we run the program, and at the execution time the modified part of the program writes the run time information we need into the Log.

The second step represents the value-less execution. Here, dependencies are analyzed with the help of Log data and, as a result, provenance of the program output is computed.

In the next sections, instrumentation and dependency analysis components are described in more detail.

The provenance analysis approach proposed here seems to be similar to the eager(book-keeping) approach of computing provenance [CCT09], but a detailed comparison is required, which, unfortunately, would go beyond the scope of the thesis.

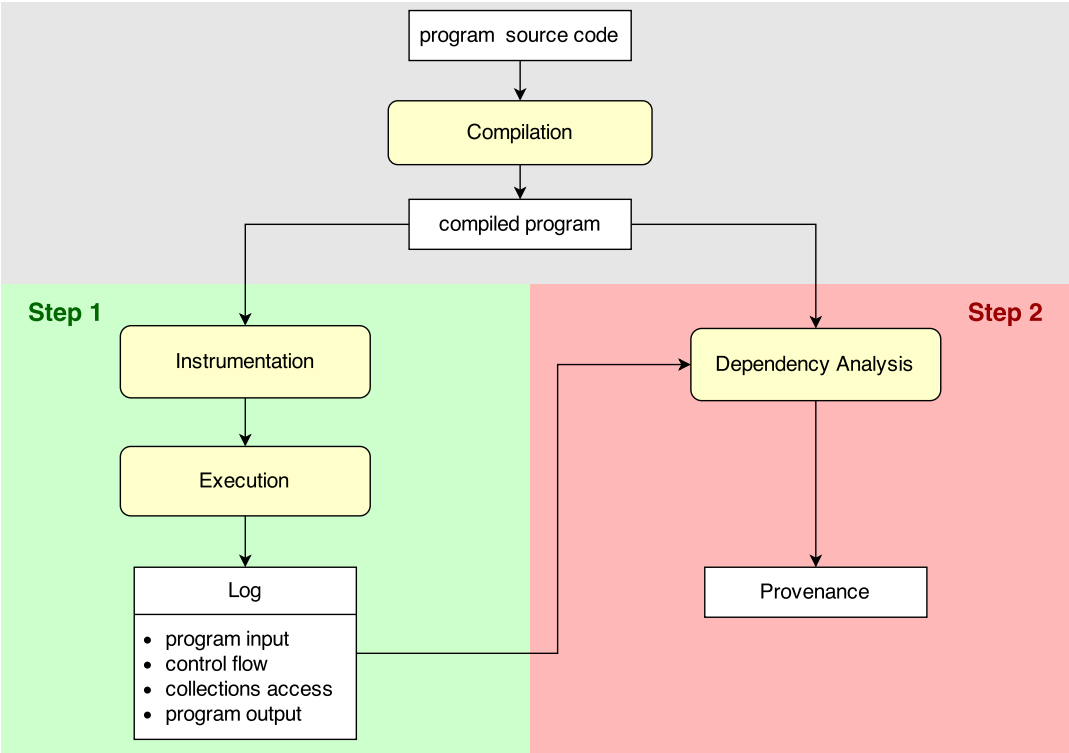


Figure 3.1: Computing provenance: Step 1 - instrumentation and execution, Step 2 - provenance analysis

3.2 Instrumentation

Instrumentation is widely used in programming. It is a set of techniques which add specific code to source or binary files in order to produce some additional information about the program while executing it. Depending on the goal of the application analysis, the additional data could be:

- performance counters, which allow applications performance measure
- debugging messages
- exception messages
- information needed for profiling or code tracing
- data logging, e.g. event logging

We instrument programs in order to log data needed to derive dependence provenance. This data are control flow decisions in form of Booleans, information about collection access in form of accessed list indices or dictionary keys and, finally, the whole input and output data sets.

Note, that the original program semantics is *not* changed by the instrumentation step, we just *add* new functionality for the specific task to be done. The primary program output is still given.

The input and the output are logged, because the goal is to derive provenance for the output and, in this manner, explain it in terms of the input.

The control flow and the subscripts logs are central, since in the analysis step we want to reproduce the program flow at the execution time. Both are represented by a chronological stream of Boolean or subscript values respectively. To illustrate this, a pseudocode example is shown in Listing 3.1

Listing 3.1: Control flow and subscripts Log

```
1 list = [1, 2, 3, 4]    # a list containing 4 numbers
2 length = len(list)
3 i = 0          # index variable
4 while i < length:    # ctrl flow log: execute loop body? true/
    false
5   number = list[i]   # subscripts log: value of i?
6   i += 1
7   if number % 2 == 0   # ctrl flow log: execute if block? true/
      false
8     print "even"
9   else:
10    print "odd"
11
12 # output: odd, even, odd, even
13
14 # iteration      Log state (true = t, false = f)
15 #   1                ctrl flow:  [t, f,]
16 #                    subscripts: [0,]
17 #   2                ctrl flow:  [t, f, t, t,]
18 #                    subscripts: [0, 1, ]
19 #   3                ctrl flow:  [t, f, t, t, t, f,]
20 #                    subscripts: [0, 1, 2,]
21 #   4(last)          ctrl flow:  [t, f, t, t, t, f, t, t,]
22 #                    subscripts: [0, 1, 2, 3]
23 #
24 # The next while condition evaluation says "false",
25 # i.e. 'loop body is not executed' decision is logged:
26 # ctrl flow:  [t, f, t, t, t, f, t, t, f]
27 #
28 # Log state after execution finished:
29 # ctrl flow:  [t, f, t, t, t, f, t, t, f]
30 # subscripts: [0, 1, 2, 3]
```

A `list` of four numbers is iterated and for each item `"even"` is printed if the value is even and `"odd"` is printed if the value is odd.

In the execution order the `ctrl flow log` is updated with the boolean values, which `while`(line `4`) or `if`(line `7`) conditions respectively were evaluated to. In the first iteration, `while` condition was satisfied, so the first entry in the log is `true`. In contrast, the `if` condition was not satisfied, because `1` is not even. Thus, the second entry in the log is `false`. The state of the log after each

iteration is shown in lines `12-20`.

Due to the reading of list items in line `5` the value of the index variable `i` is appended to the `subscripts log`. Its state after each iteration is given too.

Summing up, in the instrumentation step we extend a given program with additional instructions, so that input, output, control flow and collection access data are written to a Log when the program is executed.

3.2.1 Execution

Instrumentation itself just extends the program. After the instrumentation the modified program has to be run. Original as well as added functionality is then executed, producing the output primarily meant in the program and, at the same time, specific output coming from the instrumented parts of the program, which is in our case the Log data.

3.3 Dependency Analysis

In the second step we derive dependence provenance through *symbolic execution* of the original program.

Symbolic execution is an analysis technique, which is not working with actual variable values but assigns symbols as variable's value instead. We adapt this concept to our needs by assigning provenance sets and not symbols.

The program is analyzed in a *static* way, that is we are not actually executing it, but follow the program flow as it would be executed if we would run it.

Each value has a dependency set associated with it. When some computations are done on the values we add or remove dependencies to or from the set according to the semantics of the computation. In this manner, after we have analyzed the whole program, dependence provenance set of the output values is computed.

When a control flow decision has to be made, i.e in a loop or in if-else blocks, we don't consider several possible flows, but check the condition's boolean value we have logged in the previous step and interpret only the branch which was really executed at run time. Consequently, the result dependence set includes only dependencies that actually contributed to the output and does not contain dependencies from program parts, which were not processed. In other words, for a value we only consider dependencies present in a dynamic slice of the program on that value.

Collection access is interpreted in the same way. We look up the index to access in our log-tables in the database and then work with the dependency provenance set associated with the collection's element at that index.

3.4 Python Bytecode Approach

This tool is supposed to get Python programs as input. Thus, the instrumentation and the analysis steps of the algorithm have to be applied to Python programs. Program analysis in general provides many techniques, some working on source code and some with binary files. In this work, we are going to instrument and analyze compiled Python code, the *bytecode*.

4 Python Bytecode

Chapter Outline

In this chapter we explain what *bytecode*, a structure Python programs are compiled to, looks like. It is more a vertical slice through the material, since the compilation process and Python language internals is a large topic. We will learn what Python code objects are and how we get a code object associated with a module or a function. Finally, we will acquaint ourself with modification of bytecode and investigate, what the bytecode representation of some code constructs looks like.

4.1 CPython

There are several alternate implementations of Python. In this work, wherever we say "Python" the CPython implementation is meant. CPython is the original and most-maintained implementation of Python, written in C [Fou13b].

All code in this thesis was written in and tested against Python 2.7.5. Furthermore, since other (compiler) versions could translate same source code statements and constructs to different bytecode representations and structures, the input programs are restricted to use of bytecode instructions as implemented for Python 2.7.5. The latter are described in the Documentation of the `dis` module.

4.2 Program Execution in Python

Similar to Java, Python does not translate source code to machine code directly, but compiles it to an intermediate result first, bytecode. It is stored and can be accessed as a .pyc file. After compilation Python's Virtual Machine executes the produced bytecode [Fou15].

The compilation chain is illustrated in the next figure:

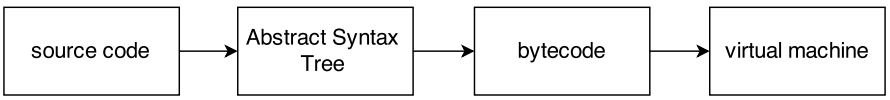


Figure 4.1: Program execution in Python

In its concept Python interpreter is similar to the microprocessor. When executing a program it is processing an instruction stream. This instruction stream is the bytecode, we can think of it as machine code for the Python Virtual Machine. For managing operands and computation results the interpreter uses a stack (in comparison to registers in a microprocessor), that is why *Python Stack Machine* is another name for the Python interpreter.

The next section shows, how the Stack Machine is working.

4.3 Bytecode

With the help of Python's *dis* module we can bring bytecode in a readable form, i.e. *disassemble* it. Disassembled bytecode looks like microprocessor instructions. A simple example of a program and its bytecode are shown below in the Listings 4.1 and 4.2 respectively. The bytecode of the program is illustrated in the disassembled form mentioned above.

Listing 4.1: Simple python program: function returns sum of two arguments

```
1 def f(a, b):
2   res = a + b
3   return res
```

Listing 4.2: Bytecode representation of the function

```
1 src line  address  opcode            oparg
2 ------------------------------------------------------------
3     2        0 LOAD_FAST                0 (a)
4              3 LOAD_FAST                1 (b)
5              6 BINARY_ADD
6              7 STORE_FAST               2 (res)
7     3       10 LOAD_FAST                2 (res)
8             13 RETURN_VALUE
```

The function's body consists of two lines. These are represented by the two bytecode blocks in the disassembled output. In `src line` column in the Listing 4.2 we see numbers `2` and `3` each introducing the block of instructions representing the source code line `2` and `3` respectively.

In most cases a single line of source code is translated to multiple primitive bytecode instructions. Indeed, from the three lines of code in our example program six lines bytecode were produced. A simple addition expression in line `2` of our sample code is translated to four bytecode operations: loading the first operand, loading the second operand, adding the operands, and finally storing the result. In general, depending on how complex is the statement to translate, the corresponding bytecode can become very long.

There is some more information in the listing we see as well:

- the `address` of the instruction represents its position in the compiled code as number of bytes from the beginning of the .pyc file (instructions are all three bytes long).
- the `opcode`, i.e. operation code name
- the `oparg`, i.e. the operand, which is then resolved to the actual instruction's argument, if the instruction requires any parameters. In parentheses we see that resolved argument.

Python Interpreter reads the instructions one by one and produces a result through executing the operation coded in the opcode on the operands. Operands and operation results are stored on an internal stack. The stack is a temporary area, where only references pointing to objects in the heap are stored. While executing an instruction Python interpreter pushes and pops the references with respect to the opcode. In other words, the interpreter creates and destroys objects in the heap according to the current instruction and pushes or pops the references to these objects to or from the stack.

Figure 4.2 below illustrates what is happening when the bytecode of the program from above is executed. There TOS means top of the stack, PC means program counter.

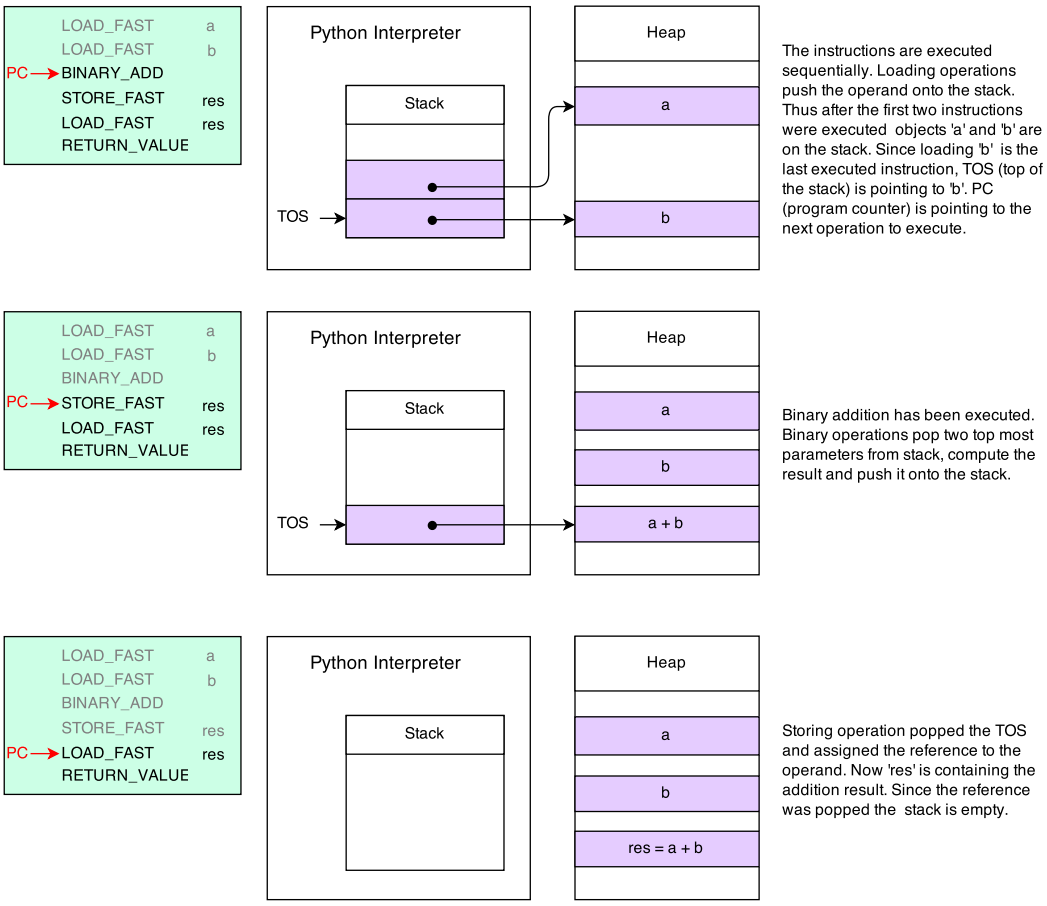


Figure 4.2: Bytecode execution

The figure does not explain how the last two instructions are executed. The last loading operation puts the reference to the result variable onto the stack again. Then, the RETURN_VALUE operation returns whatever is on TOS to the caller.

Note that binary operations are working on **two** operands and are not processing binary numbers.

The full list of opcodes and their meaning and further detailed information to bytecode is described in Python Documentation to bytecode.

4.4 Program Code Modification

Python makes it easy to instrument programs, since it is possible to modify compiled bytecode directly and execute this code afterwards. To understand, how exactly bytecode can be modified, we need to know that Python bytecode is represented by code objects.

4.4.1 Python Code Objects

In Python everything is an object [Fou13a]. This means, bytecode is an object too. For each module, class, function or an interactively typed command there is a single *code object* associated with it and representing its bytecode. For instance, for a function we get the code object of the function's body by reading function's attribute *__code__*. Code objects have the type "CodeType" and like the other objects are stored in the heap.

In addition to the bytecode itself a code object also stores several attributes, which we do not discuss in detail here and refer to the section describing code objects of the Python Documentation instead [Fou13a]. At this point we only put on record that the opargs are resolved by looking up the *co_names*, *co_varnames* and *co_consts* attributes of a code object, which are tuples containing the names of global variable names, local variable names (including function arguments) and constants respectively. But even this resolution in effect invoke checking other code object attributes too, so we do not look at it in detail.

We illustrate this with the sample code from the previous section.

Listing 4.3: Code object attributes: variables and constants

```
1  def f(a, b):
2    res = a + b
3    return res
4
5  print"co_varnames: " , f.__code__.co_varnames
6  print"co_names: " , f.__code__.co_names
7  print"co_consts: " , f.__code__.co_consts
8
9  Output:
10 co_varnames:  ('a', 'b', 'res')
11 co_names:  ()
12 co_consts:  (None,)
```

Since no global variables are used the `co_names` tuple is empty. But if wee consider a slightly different function as in Listing 4.4, we see that now the global variable `c` is included in the tuple of global variables of the function.

Listing 4.4: Code object attributes: global variables

```
c = 30

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

print"co_varnames: " , f.__code__.co_varnames
print"co_names: " , f.__code__.co_names
print"co_consts: " , f.__code__.co_consts

Output:
co_varnames:  ('a', 'b', 'res')
co_names:  ('c',)
co_consts:  (None,)
```

4.4.2 Nested Code Objects

Whenever a *block*, which is a piece of Python program text that is executed as a unit, is compiled, a code object is created. Since it is possible to define nested blocks, e.g. a module with a function definition in it, it is also possible to get nested code objects. Here it is important to know that code objects are immutable. Therefore, if we want to know whether some code object contains more code objects, we have to look for entries of the "CodeType" type in its *co_consts* tuple.

Consider the following code snippet.

Listing 4.5: Simple module code

```
a = 10
b = 20
res = a + b
print res
```

Listing 4.6: Variables and constants of the code

```
co_varnames:  ()
co_names:  ('a', 'b', 'res')
co_consts:  (10, 20, None)
```

We save this piece of code as a file "someModuleExample.py". From now we concentrate on cases, where the input Python program is given as .py file, since that is the input interface defined in our tool. Getting the input in that way we have to make use of some Python modules to get the code object associated with the program in file, namely *py_compile* and *marshal*. Following is the code getting a code object from a source code file:

Listing 4.7: Code object attributes: global variables

```
# compile the source code, i.e. create the code object and
    save it to a .pyc file
py_compile.compile("someModuleExample.py")
# open compiled code
```

```
4   fd = open("someModuleExample.pyc", 'rb')
5   # first 8 chars of the stream are representing meta data
6   magic = fd.read(4) # python version specific magic number
7   date = fd.read(4)  # compilation date
8   # now the code object itself is the next in the stream,
9   # load it
10  code_object = marshal.load(fd)
11  # reference to the code object in heap is enough, close file
12  fd.close()
```

In Listing 4.6 variables and constants of the loaded code object are shown. Here it is worth pointing out that at module level all variables are global. Accordingly to that `co_varnames` tuple is empty, as we see. Insofar there are no other points to mention to this code object.

Now, suppose we have extracted the addition to a function. For convenience, we also rename the module level variables:

Listing 4.8: Nested code blocks: function definition within a module block

```
1   def f(a, b):
2       res = a + b
3       return res
4
5   x = 10
6   y = 20
7   res = f(x,y)
8   print res
```

This code piece we will save in a different file "someFunctionExample.py". We load its code object the same way as above and look at the same attributes:

Listing 4.9: Nested code blocks: function's code object referenced as constant by the enclosing module code object

```
1   co_varnames:  ()
2   co_names:  ('f', 'x', 'y', 'res')
3   co_consts:  (<code object f at 01FFB608, file
        "someFunctionExample.py", line 1>, 10, 20, None)
```

Whereas the code piece in Listing 4.5 was a single block, here the function definition is an extra block in the module code. We are still observing the module code object, but now it has the function's name in its global names tuple and the reference to function's code object in its constants tuple. Additionally, function code object's address in the memory and function definition's position in the source code are given. If we would define more functions in this module, more code objects would be listed in the constants tuple of the module code object.

Finally, we present a recursive function, which we can use to explore (nested) code objects structure of any given file:

Listing 4.10: Explore code object's structure

```
import types

def explore_code_object(co_obj, indent=''):
    print indent, co_obj.co_name,  "(lineno:",
        co_obj.co_firstlineno, ")"
    for c in co_obj.co_consts:
        if isinstance(c, types.CodeType):
            explore_code_object(c, indent + '\t')

# Applied to "someModuleExample.py":
<module> (lineno: 1 )

# Applied to "someFunctionExample.py":
<module> (lineno: 1 )
  f (lineno: 1 )
```

4.4.3 Byteplay

There are many frameworks for simplified bytecode visualisation and analysis. In this work we use byteplay.

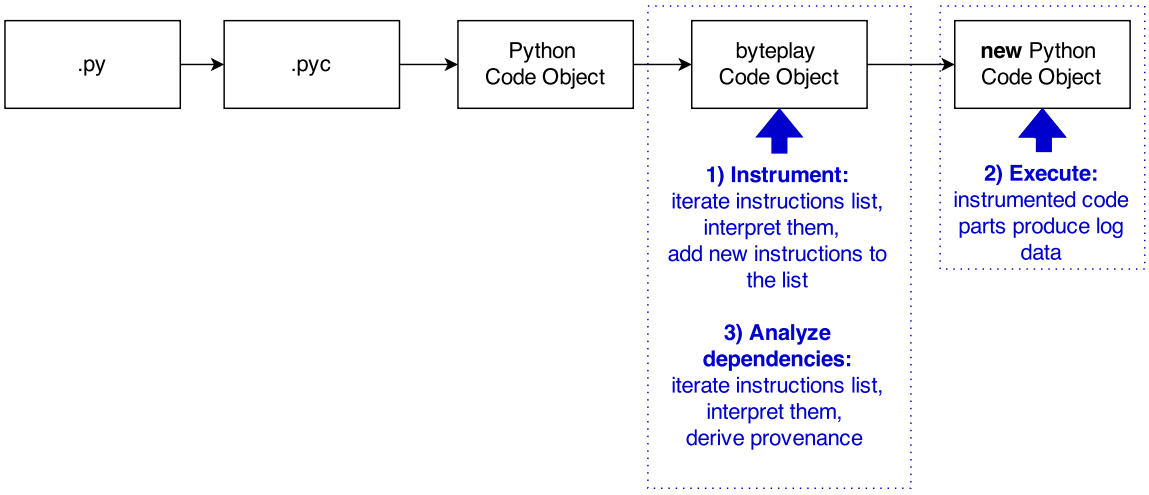


Figure 4.3: Using byteplay for bytecode analysis

Byteplay provides a wrapper object for Python code objects. The bytecode is then represented by a list of instructions, which in their turn are represented as tuples `(opcode, oparg)`.

4.4.4 Modifying Bytecode

The most interesting code object's attribute for us is the *co_code*. It represents the sequence of the code object's bytecode instructions. Unfortunately, *co_code* is a read only attribute, so we can not modify it directly. However, for our purposes we don't need to. In terms of our concept, it is enough to execute the extended bytecode just once, that is we don't have to make the changes persistent. Thus, we just create a new code object, whose *co_code* attribute contains a modified copy of the existing instructions sequence of the given code object. This new code object can be consequently run and will execute all the additional effects.

We illustrate the idea on the example from Listing 4.5. The disassembled bytecode of the program looks like what follows:

Listing 4.11: Bytecode of the small sample in Listing 4.5

```
1 src line  address  opcode       oparg
2 ------------------------------------------------------------
3   1         0 LOAD_CONST         0 (10)  # 10 loaded on TOS
4             3 STORE_NAME         0 (a)   # 10 stored into a
5                                          # here empty stack again
6   2         6 LOAD_CONST         1 (20)  # 20 loaded on TOS
7             9 STORE_NAME         1 (b)   # 20 stored into b
8                                          # here empty stack again
9   3        12 LOAD_NAME          0 (a)   # a's value loaded
10           15 LOAD_NAME          1 (b)   # b's value loaded
11           18 BINARY_ADD         # two values popped, sum on TOS
12           19 STORE_NAME         2 (res) # TOS stored into res
13                                         # here empty stack again
14  4        22 LOAD_NAME          2 (res)  # res's value on TOS
15           25 PRINT_ITEM                  # TOS popped and printed
16           26 PRINT_NEWLINE               # line break printed
17           27 LOAD_CONST         2 (None)  # None on TOS
18           30 RETURN_VALUE         # TOS returned to the caller
```

Suppose we want to print a string after `a` was initialized. For that we add a few instructions (in green) right after the bytecode block of the line `1`. In fact we add the same instructions as used at the end of the program, where `res` is printed:

Listing 4.12: Modified bytecode

```
src line  address  opcode                oparg
-------------------------------------------------------------
  1         0 LOAD_CONST                 0 (10)
            3 STORE_NAME                 0 (a)
            6 LOAD_CONST                  'awesome'
            9 PRINT_ITEM
           10 PRINT_NEWLINE

  2        11 LOAD_CONST                 1 (20)
           14 STORE_NAME                 1 (b)

  3        17 LOAD_NAME                  0 (a)
           20 LOAD_NAME                  1 (b)
           23 BINARY_ADD
           24 STORE_NAME                 2 (res)

  4        27 LOAD_NAME                  2 (res)
           30 PRINT_ITEM
           31 PRINT_NEWLINE
           32 LOAD_CONST                 2 (None)
           35 RETURN_VALUE
```

We execute the new created code object and get:

Listing 4.13: Execute modified bytecode

```
exec newCodeObject

# Output:
awesome
30
```

In Python all code objects return something, even if the programmer didn't define any `return` statements in module's or function's code. If no `return` statement was defined `None` is still returned. Thus, the `RETURN_VALUE` instruction is always the last instruction of a code object.

5 Implementation

Chapter Outline

This Chapter is devoted to the implementation of the tool.

In Section 5.1 a brief overview of the application components is given.

Section 5.2 describes the *instrumenter*. This component is responsible for the first step of the algorithm proposed in Chapter 3, i.e instrumentation and execution of the input program in order to get the Log.

Section 5.3 present the *analyzer* component, which implements the symbolic execution of the input program and computes the dependence provenance.

5.1 Application Overview

The tool is composed of two components: *instrumenter* and *analyzer*, see Figure 5.1.

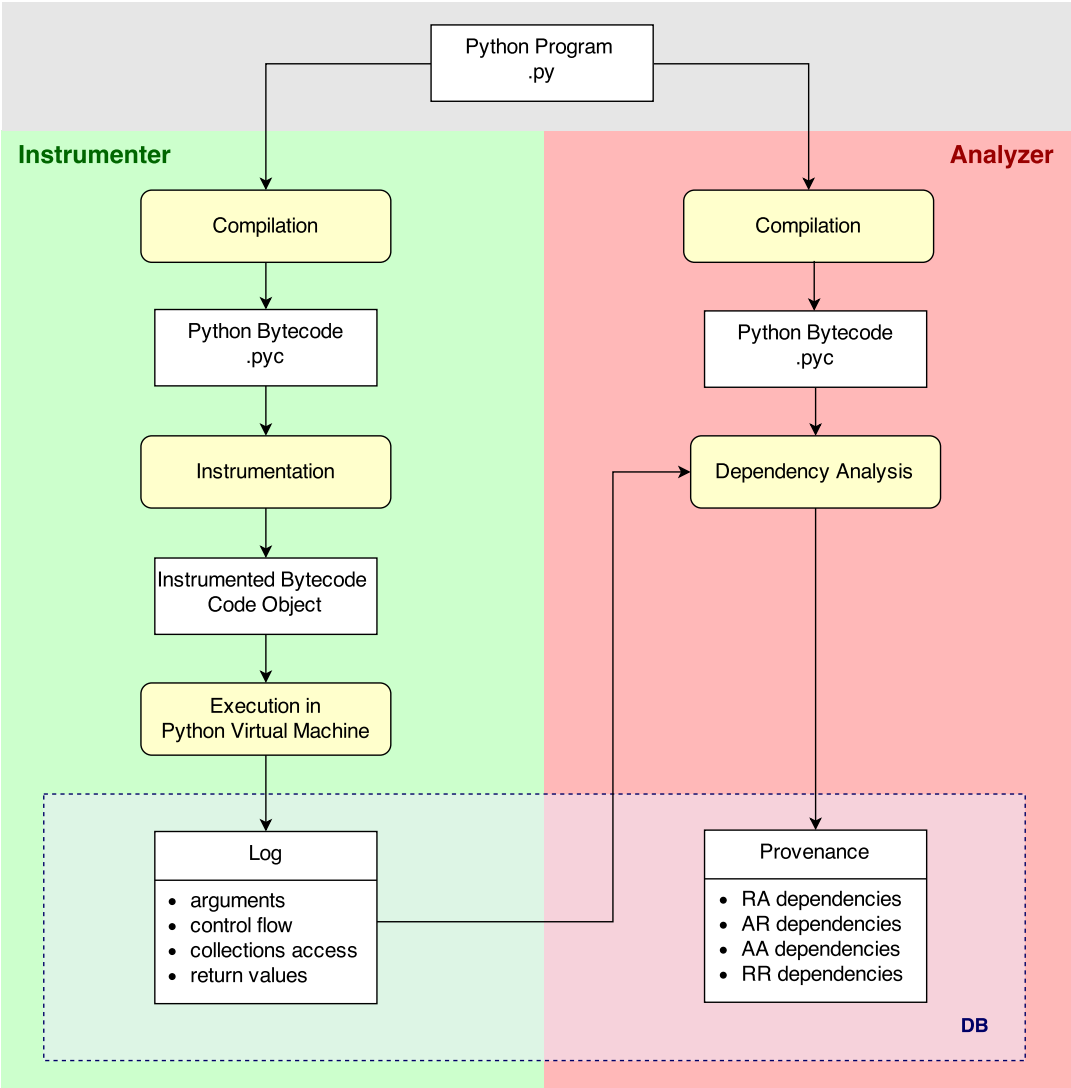


Figure 5.1: Application workflow

The **Instrumenter** is responsible for step one in the algorithm 3.1, i.e. getting the Log through code instrumentation and execution, whereas the **Analyzer** proceeds the second step, thus, derives dependence provenance for the program output with the help of the logged data.

Both components must be executed separately with the input program given as an argument. The workflow is then as follows.

The instrumenter compiles the program, instruments the bytecode, executes the modified bytecode, whereat Log data is extracted, and after execution writes the data to the Log. The Log is stored in a database.

As next, the analyzer reads the Log, compiles the program, interprets the bytecode with respect to the logged run time data and computes provenance sets for program output items. Finally, derived dependencies are stored in the database as well.

5.1.1 Program Interface

Database

For the Log and as a permanent dependence provenance storage we made use of a PostgreSQL 9.3.5 database. To communicate with it Psychopg is used as a database adapter for the Python language.

Setup and Run

For application setup and 'how to' see "ReadMe.txt" in the project directory.

Visualisation Tool

To visualize computed dependencies the already mentioned web-based tool [Bet14] can be used.

Note that relation and record dependencies can not be displayed by the tool, just the field dependencies (compare [Che07]).

5.2 Instrumentation

Instrumenter's job is the *recording* of run time data needed for provenance analysis, so that we can access it after the program was executed. In implementing this, several components emerged, which are illustrated in the UML Diagram 5.2.

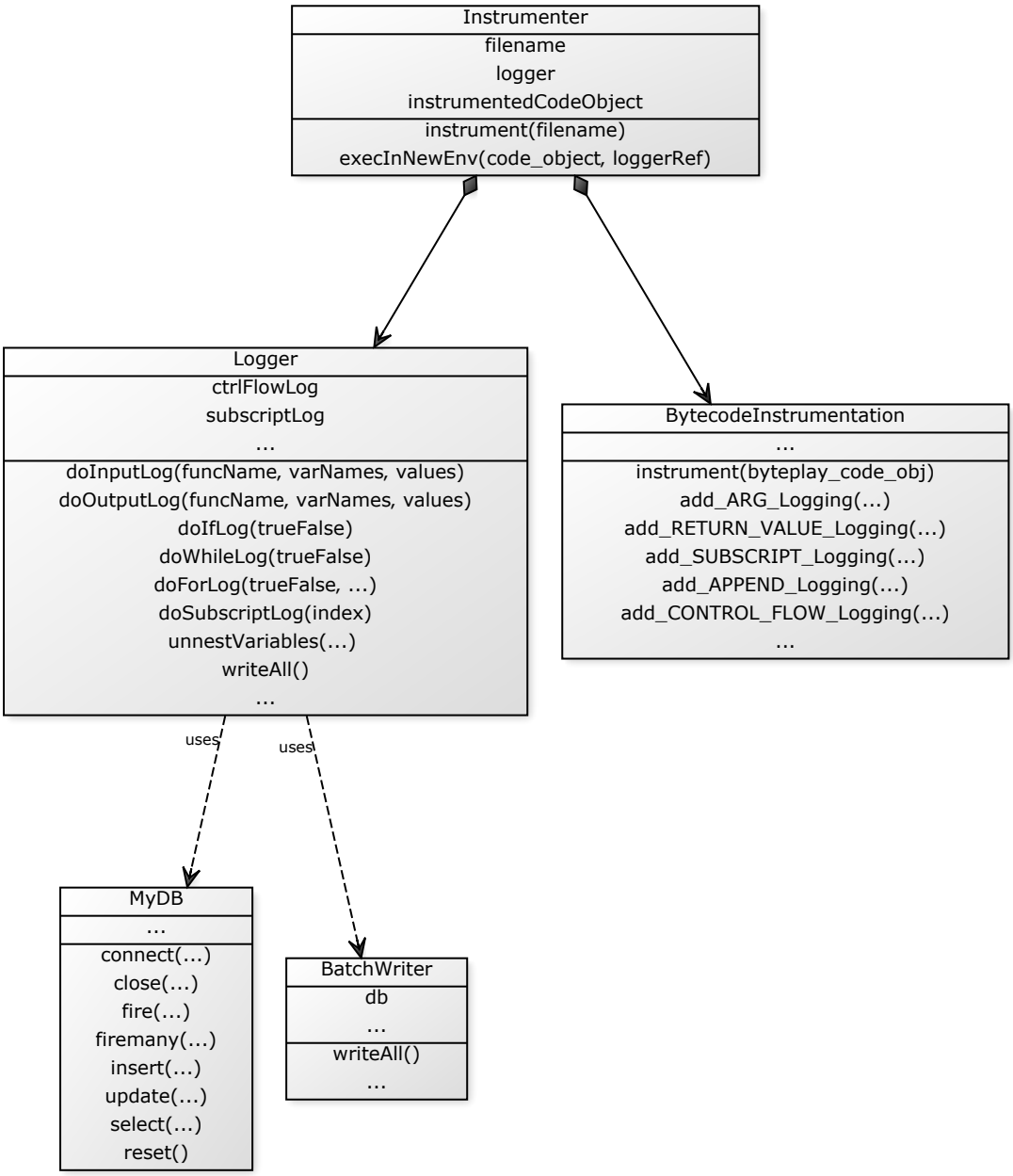


Figure 5.2: Class and module structure of the instrumenter component.

The class Logger provides functionality to store the data.

BytecodeInstrumentation module, as the name implies, implements the instrumentation of the program extending it with the logging behaviour. It modifies the input program so that it knows about the Logger. During the execution the program calls Logger's data saving methods with appropriate information assigned to the arguments. See Listing 5.1 for an example:

Listing 5.1: Before instrumentation

```
def someFunction( argument1, argument2):
# arguments must be logged

  if someCondition:     # ctrl flow must be logged
    ... # make some computation
  else:
    ... # make another computation

  return result     # return value must be logged
```

Listing 5.2: After instrumentation

```
def someFunction( argument1, argument2):

  # log argument values:
  logger.doInputLog(argument1, argument2)

  # log ctrl flow:
  logger.doIfLog(someCondition)
  if someCondition:
    ... # make some computation
  else:
    ... # make another computation

  # log return value:
  logger.doOutputLog(result)
  return result
```

These two components are implementing the main functionality of the instrumenter. Other classes, MyDB and BatchWriter, are support classes for the Logger and are needed for communication with the database.

5.2.1 Logger

The class Logger provides an *interface* to store the needed run time information.

After instrumentation the input program logs its own run time data by using this interface. For that to work, an instance of the Logger class is passed to the input program as part of the global namespace in which the program is

executed, see Listing 5.3. In this manner, the program can access the Logger object at run time, as if it would be defined by the program itself.

Listing 5.3: Passing a Logger object reference to the executed program

```
# create a new Logger instance:
loggerRef = Logger.Logger()
# create a namespace, which knows about this Logger instance:
newNameSpace = {'logger':loggerRef}
# execute instrumented code in this namespace,
# which provides access to the Logger object:
exec code_object in newNameSpace
# at run time newNameSpace is code_object's global namespace
# and, thus,
# code_object can access one more variable - 'logger',
# which is referencing the same object as loggerRef
```

Then, while running, the input program calls Logger's `doSomeLog(...)` methods, which remember the data by saving it in the heap.

to appropriate lists about program's input and output, control flow, collection access, namespaces and function calls. Lists are kept as *logs* for each of the different information kinds. When logging arguments and return values a bit more work has to be done in view of retaining in the database. Here, the `unnestVariables()` function brings structured values like lists and dictionaries into an unnested form so that each value, that is, the collection itself as well as each of its elements, is represented by a table entry with a unique id.

After the program execution has finished, the Logger object still lives and we can still access it as `loggerRef`, see Listing 5.4:

Listing 5.4: After execution

```
loggerRef = Logger.Logger()
newNameSpace = {'logger':loggerRef}
exec code_object in newNameSpace

# here, the input program execution is finished,
# run time data is stored internally in the Logger object
# so we can write it to the database:
loggerRef.writeAll()
```

In this last step we write the Log into the database.

Now, what is needed is a way to place Logger's `doSomeLog(...)` methods calls into the code of the input program. BytecodeInstrumentation takes care of this.

5.2.2 Bytecode Instrumentation

BytecodeInstrumentation module upgrades a given program to log its own run time data. This is done as follows.

All code objects defined in the input module are extracted and then instrumented one by one. As already mentioned, we use byteplay to simplify the bytecode manipulation in technical terms 4.4.3. So what we are working with is a list of bytecode instructions each represented by a tuple of operation code and operation argument `(opcode, oparg)`.

Calling Logger methods

The list of bytecode instructions is processed pretty straight forward. We iterate over the instructions and catch opcodes which are significant parts of an `if` statement, a `while` or a `for` loop, some kind of collection access or the return statement. Then we extend the list at the position where such an instruction was identified, by placing a call to an appropriate Logger method with the Log data assigned to the arguments. This is always done in the same manner:

Listing 5.5: Bytecode block calling to some method in the Logger class

```
1 # code before instrumentation part
2   ...
3 LOAD_GLOBAL      logger     # logger instance loaded on TOS
4 LOAD_ATTR        doSomeLog  # TOS repl. by method's code object
5 LOAD_FAST        someArg_1  # now load all method arguments
6   ...
7 LOAD_FAST        someArg_n
8 CALL_FUNCTION    n          # finally, the method is executed
9 POP_TOP   # remove method's return value, no use for it here
10  ...
11 # code after instrumentation part
```

Stack Invariant

An important issue we must pay attention to when instrumenting programs is that the primary program flow must stay uninfluenced. In particular, Python's stack state before execution of some instrumented part must be exactly the same after execution of this instrumented part.

For example, all Python functions push some return value onto the stack [1], that is, also functions we added through the instrumentation. But the original program code does not expect these additional return values to be on the stack. It has no use for it and so, it will not work if the values stay there. Consequently, we must always delete the return value of the Logger functions, see `POP_TOP` instruction in line 9 in the previous Listing 5.5.

[1] See an earlier section 4.4.4

In general, all values additionally loaded to the stack by instrumented code parts must be also popped at some point before the primary code execution proceeds. On the other hand, naturally, no values *needed* by the original program must be changed or missed.

How to get the arguments: DUP_TOP and ROT_TWO opcodes

Often the values that should be passed to the Logger are already on the stack. In these cases we duplicate the value on the stack by the DUP_TOP instruction, see line 4 in Listing 5.6.

Thereafter, we have two possibilities:

- either save the value to a variable and load it later as an argument:

Listing 5.6: Save argument value in a variable

```
1   ...
2 # through loading or some computation
3 # the needed value is now at the top of the stack
4 DUP_TOP                          # value loaded one more time
5 STORE_FAST            someArg_1  # someArg_1 contains the value
6 # here, stack state same as before DUP_TOP
7   ...
8 LOAD_GLOBAL           logger
9 LOAD_ATTR             doSomeLog
10 LOAD_FAST            someArg_1  # the value passed as argument
11  ...
12 LOAD_FAST            someArg_n
13 CALL_FUNCTION        n
14 POP_TOP
15  ...
16 # code after instrumentation part
```

- or some of the stack items moving instructions is used [2]:

Listing 5.7: Rotate stack items to move the needed value to the right position as method's argument

```
1   ...
2 # through loading or some computation
3 # the needed value is now at the top of the stack
4 DUP_TOP                          # value at TOS here
5 LOAD_GLOBAL           logger
6 LOAD_ATTR             doSomeLog   # method's object at TOS here
7 ROT_TWO                           # value at TOS again
8 LOAD_FAST             someArg_2
9   ...
10 LOAD_FAST            someArg_n
```

[2] See ROT_... instructions in the opcodes list

```
11 CALL_FUNCTION        n
12 POP_TOP
13   ...
14 # code after instrumentation part
```

However, to know which of Logger's methods is to call when, we have to know, how Python's respective data and control structures are represented in bytecode. This is what the next section is about.

It is worth noting, that indeed *all* code objects are instrumented, even when some of them represent functions which are never called. An objection can be raised, that instrumenting all functions in the module can reduce performance and only called functions should be instrumented. However, cases where uncalled functions are present in the input are rather unlikely for translated SQL queries - the main use case of the tool, and the implemented approach was proved good enough while testing. Though, the issue could be still considered as an optimization possibility.

5.2.3 What has to be logged

As already mentioned, for dependency analysis we need the following run time information:

- input data, i.e. argument values of called functions
- output data, i.e. return values of called functions
- collections access information, i.e subscripts of all reading and writing operations on collection elements, such as list, tuple or dictionary items
- control flow information, i.e a sequence of boolean values representing chronological record of decisions, whether an `if` block or a loop body was executed

In what follows, correspondent Python code constructs, their bytecode representation and its instrumented version are presented.

In all the listings in this section, bytecode structures representing relevant code constructs are colored red. Code blocks added by the instrumentation module are colored blue.

Input

Instrumentation of the program to log its arguments is done by the `add_ARG_Logging(...)` function.

Consider a function as in Listing 5.8. It has three arguments `a, b, c`.

Listing 5.8: Log function arguments

```
def inputInstrumentationSample(a, b, c):
    print "function's body starts here"
    print a,b,c
```

We want the function to log its arguments, so its instrumented version would do something like the following:

Listing 5.9: Instrumented source code version

```
def inputInstrumentationSample(a, b, c):

  # the logging part:
  PBIM_names = ['a', 'b', 'c']  # The Logger needs args names
  PBIM_values = [a, b, c]       # and their values
  # now use Logger's input logging method:
  # doInputLog(funcName, varNames, values)
  logger.doInputLog('inputInstrumentationSample', \
                    PBIM_names,PBIM_values)

  # the primary part:

  print "function's body starts here"
  print a,b,c
```

The bytecode of the primary program is illustrated in Listing 5.10. As we already know, the numbers 2 and 3 at the left are introducing the bytecode of source code line 2 and 3 respectively.

Listing 5.10: Before instrumentation

```
2          1 LOAD_CONST           "function's body starts here"
           2 PRINT_ITEM
           3 PRINT_NEWLINE

3          5 LOAD_FAST            a
           6 PRINT_ITEM
           7 LOAD_FAST            b
           8 PRINT_ITEM
           9 LOAD_FAST            c
          10 PRINT_ITEM
          11 PRINT_NEWLINE
          12 LOAD_CONST           None
          13 RETURN_VALUE
```

The bytecode of the instrumented version is shown in the next Listing 5.11. The logging part is represented by instructions in lines 1 - 17 and the primary part in the rest of the listing.

Listing 5.11: After instrumentation

```
1            0 LOAD_CONST           'a'
2            1 LOAD_CONST           'b'
3            2 LOAD_CONST           'c'
4            3 BUILD_LIST           3
5            4 STORE_FAST           PBIM_names
6            5 LOAD_FAST            a
7            6 LOAD_FAST            b
8            7 LOAD_FAST            c
9            8 BUILD_LIST           3
10           9 STORE_FAST           PBIM_values
11          10 LOAD_GLOBAL          logger
12          11 LOAD_ATTR            doInputLog
13          12 LOAD_CONST           'inputInstrumentationSample'
14          13 LOAD_FAST            PBIM_names
15          14 LOAD_FAST            PBIM_values
16          15 CALL_FUNCTION        3
17          16 POP_TOP

19 2         1 LOAD_CONST           "function's body starts here"
20           2 PRINT_ITEM
21           3 PRINT_NEWLINE
22
23 3         5 LOAD_FAST            a
24           6 PRINT_ITEM
25           7 LOAD_FAST            b
26           8 PRINT_ITEM
27           9 LOAD_FAST            c
28          10 PRINT_ITEM
29          11 PRINT_NEWLINE
30          12 LOAD_CONST           None
31          13 RETURN_VALUE
```

This part was easy. But how do we know which arguments a function has, when we don't *see* the signature? Or what the function's name is? The answer is: from the code object. The `name` property of function's code object saves the function's name and the `args` attribute contains names of all function arguments [3]. Hence, the instrumenter iterates over this list and add arguments logging bytecode to the input code object, see Listing 5.12

Listing 5.12: Arguments instrumentation

```
1 loggingPart = []
2 numArgs = len(code_object.args)
3 # creates the bytecode in lines 1 - 5:
4 for arg in code_object.args:
5   # load arg name <-> string <-> const:
6   loggingPart.append((LOAD_CONST, arg))
7 loggingPart.append((BUILD_LIST, numArgs ))
```

[3]Note that we assume functions not to use `*args`, `**kwargs`

```
8 loggingPart.append((STORE_FAST, 'PBIM_names'))
9 # creates the bytecode in lines 6 - 17
10 for arg in code_object.args:
11   # load arg value:
12   loggingPart.append((LOAD_FAST, arg))
13 loggingPart.extend([(BUILD_LIST, numArgs ),
14                      (STORE_FAST, 'PBIM_values'),
15                      (LOAD_GLOBAL, 'logger'),
16                      (LOAD_ATTR, 'doInputLog')
17                      (LOAD_CONST, codeObject.name),
18                      (LOAD_FAST, 'PBIM_names'),
19                      (LOAD_FAST, 'PBIM_values'),
20                      (CALL_FUNCTION, 3),
21                      (POP_TOP, None)])
22 # now add the logging bytecode into original instructions list:
23 code_object.code[0:0] = loggingPart
```

Function Calls

Logger's `doInputLog(...)` method also implements logging of function calls. Function's name is passed as one of the arguments.

Output

In Python all code objects return something, even if the programmer didn't define any `return` statements in module's or function's code [4]. If no `return` statement was defined `None` is still returned. Thus, the `RETURN_VALUE` instruction is always the last instruction of a code object.

As an example, function in Listing 5.13 returns a list. Again, bytecode and the instrumented result follow.

Listing 5.13: Log function return value

```
1 def outputInstrumentationSample(a, b, c):
2     res = [a,b,c]
3     print "return stmt comes next"
4     return res
```

Listing 5.14: Before instrumentation

```
1 2          1 LOAD_FAST            a
2            2 LOAD_FAST            b
3            3 LOAD_FAST            c
4            4 BUILD_LIST           3
5            5 STORE_FAST           res
```

[4] See an earlier section 4.4.4

```
6
7 3          7 LOAD_CONST           'return stmt comes next'
8            8 PRINT_ITEM
9            9 PRINT_NEWLINE
10
11 4        11 LOAD_FAST            res
12          12 RETURN_VALUE
```

Listing 5.15: After instrumentation

```
1 2          1 LOAD_FAST            a
2            2 LOAD_FAST            b
3            3 LOAD_FAST            c
4            4 BUILD_LIST           3
5            5 STORE_FAST           res
6
7 3          7 LOAD_CONST           'return stmt comes next'
8            8 PRINT_ITEM
9            9 PRINT_NEWLINE
10
11 4        11 LOAD_FAST            res

12          12 DUP_TOP
13          13 STORE_FAST           PBIM_res_value
14          14 LOAD_CONST           'res'
15          15 STORE_FAST           PBIM_res_name
16          16 LOAD_GLOBAL          logger
17          17 LOAD_ATTR            doOutputLog
18          18 LOAD_CONST           'outputInstrumentationSample'
19          19 LOAD_FAST            PBIM_res_name
20          20 LOAD_FAST            PBIM_res_value
21          21 CALL_FUNCTION        3
22          22 POP_TOP

23          12 RETURN_VALUE
```

Collections access

Fortunately, reading and writing items to tuples, lists and dictionaries works the same way in bytecode and, hence, can be instrumented in the exact same manner. We illustrate the proceeding on a list, see 5.16.

Listing 5.16: Log list items access

```
1 def collectionsInstrumentationSample(aList, index):
2     res = aList[index]   # read item
3     aList[index] = 0     # write item
4     aList.append(res)    # add an item at the end of the list
5     return res
```

The following bytecode represents the *reading* access:

Listing 5.17: Before instrumentation

```
2          1 LOAD_FAST          aList
           2 LOAD_FAST          index
           3 BINARY_SUBSCR
           4 STORE_FAST         res
```

Listing 5.18: After instrumentation

```
2          1 LOAD_FAST          aList
           2 LOAD_FAST          index

           3 DUP_TOP
           4 LOAD_GLOBAL        logger
           5 LOAD_ATTR          doSubscriptLog
           6 ROT_TWO
           7 LOAD_CONST         'lineNo'
           8 CALL_FUNCTION      2
           9 POP_TOP

          10 BINARY_SUBSCR
          11 STORE_FAST         res
```

Now *writing* of an item:

Listing 5.19: Before instrumentation

```
3          6 LOAD_CONST         0
           7 LOAD_FAST          aList
           8 LOAD_FAST          index
           9 STORE_SUBSCR
```

Listing 5.20: After instrumentation

```

3         13 LOAD_CONST         0
          14 LOAD_FAST          aList
          15 LOAD_FAST          index

          16 DUP_TOP
          17 LOAD_GLOBAL        logger
          18 LOAD_ATTR          doSubscriptLog
          19 ROT_TWO
          20 LOAD_CONST         'lineNo'
          21 CALL_FUNCTION      2
          22 POP_TOP

          23 STORE_SUBSCR
```

Finally, *appending* of an item:

Listing 5.21: Before instrumentation

```
4          11 LOAD_FAST            aList
           12 LOAD_ATTR            append
           13 LOAD_FAST            res
           14 CALL_FUNCTION        1
           15 POP_TOP
```

Listing 5.22: After instrumentation

```
4          25 LOAD_FAST            aList

           26 DUP_TOP
           27 LOAD_GLOBAL          logger
           28 LOAD_ATTR            doAppendLog
           29 ROT_TWO
           30 LOAD_CONST           'lineNo'
           31 CALL_FUNCTION        2
           32 POP_TOP

           33 LOAD_ATTR            append
           34 LOAD_FAST            res
           35 CALL_FUNCTION        1
           36 POP_TOP
```

Control Flow

Following control flow constructs are logged: `if` statements, `while` loops and `for` loops. In case of `if` statements and `while` loops whatever `if`'s or looping condition respectively was evaluated to is appended to the Log. Since usage of `for` loops in the input programs is restricted to iterating over collections, we log `True` while iterating and `False` when all items were processed.

Logging of `if` *statements* is explained on the function in the next Listing:

Listing 5.23: Log if statement

```
def ifInstrumentationSample1(someBoolean):
    if someBoolean:
        print "if block"
    else:
        print "else block"
    print "code after if stmt"
```

Note that the `LOAD_FAST` instruction in the first line could be traded off against any computation block depending on what kind of code implements the condition. The `JUMP_FORWARD` instructions is replaced by the `JUMP_ABSOLUTE`, when

a jump to some earlier point in the program is needed, for instance, when the relevant `if` construct is enclosed by a loop.

Listing 5.24: Before instrumentation

```
2          1 LOAD_FAST            someBoolean
           2 POP_JUMP_IF_FALSE    to 10

3          4 LOAD_CONST           'if block'
           5 PRINT_ITEM
           6 PRINT_NEWLINE
           7 JUMP_FORWARD         to 15

# note that line 4 containing 'else:' is omitted

5     >> 10 LOAD_CONST           'else block'
          11 PRINT_ITEM
          12 PRINT_NEWLINE

6     >> 15 LOAD_CONST           'code after if stmt'
          16 PRINT_ITEM
          17 PRINT_NEWLINE
          18 LOAD_CONST           None
          19 RETURN_VALUE
```

Listing 5.25: After instrumentation

```
2          1 LOAD_FAST            someBoolean

           2 DUP_TOP
           3 LOAD_CONST           'lineNo'
           4 ROT_TWO
           5 LOAD_GLOBAL          logger
           6 LOAD_ATTR            doIfLog
           7 ROT_THREE
           8 CALL_FUNCTION        2
           9 POP_TOP

          10 POP_JUMP_IF_FALSE    to 17

3         11 LOAD_CONST           'if block'
          12 PRINT_ITEM
          13 PRINT_NEWLINE
          14 JUMP_FORWARD         to 22

5     >> 17 LOAD_CONST           'else block'
          18 PRINT_ITEM
          19 PRINT_NEWLINE

6     >> 22 LOAD_CONST           'code after if stmt'
          23 PRINT_ITEM
          24 PRINT_NEWLINE
```

```
24          25 LOAD_CONST           None
25          26 RETURN_VALUE
```

Logging of `while` *loops* is very similar to that of `if` statements.

Listing 5.26: Log while loop

```
1 def whileInstrumentationSample(a, b, c):
2     while c > 0:
3         print("loop condition satisfied" )
4     print("code after while stmt" )
```

Listing 5.27: Before instrumentation

```
1 2          1 SETUP_LOOP           to 16
2       >> 3 LOAD_FAST             c
3          4 LOAD_CONST            0
4          5 COMPARE_OP            >
5          6 POP_JUMP_IF_FALSE     to 13
6
7 3          8 LOAD_CONST           'loop condition satisfied'
8          9 PRINT_ITEM
9         10 PRINT_NEWLINE
10        11 JUMP_ABSOLUTE         to 3
11      >> 13 POP_BLOCK
12
13 4    >> 16 LOAD_CONST           'code after while stmt'
14        17 PRINT_ITEM
15        18 PRINT_NEWLINE
16        19 LOAD_CONST            None
17        20 RETURN_VALUE
```

Listing 5.28: After instrumentation

```
1 2          1 SETUP_LOOP           to 24
2       >> 3 LOAD_FAST             c
3          4 LOAD_CONST            0
4          5 COMPARE_OP            >

5          6 DUP_TOP
6          7 LOAD_CONST            'lineNo'
7          8 ROT_TWO
8          9 LOAD_GLOBAL           logger
9         10 LOAD_ATTR             doWhileLog
10        11 ROT_THREE
11        12 CALL_FUNCTION         2
12        13 POP_TOP

13        14 POP_JUMP_IF_FALSE     to 21
14
15 3      16 LOAD_CONST            'loop condition satisfied'
```

```
16          17 PRINT_ITEM
17          18 PRINT_NEWLINE
18          19 JUMP_ABSOLUTE        to 3
19      >> 21 POP_BLOCK
20
21 4    >> 24 LOAD_CONST            'code after while stmt'
22          25 PRINT_ITEM
23          26 PRINT_NEWLINE
24          27 LOAD_CONST           None
25          28 RETURN_VALUE
```

`for` *loops* are special in the way, that collection access must be logged too, since a collection is being iterated. Though, this is not done through instrumentation, but by the Logger's `doForLog(...)` method internally.

Listing 5.29: Log for loop

```
1 def forInstrumentationSample(aList):
2     for number in aList:
3         print("iterating over the list:" )
4         print number
5     print("all list items processed" )
```

Listing 5.30: Before instrumentation

```
1 2          1 SETUP_LOOP           to 20
2            2 LOAD_FAST            aList
3            3 GET_ITER
4        >> 5 FOR_ITER             to 17
5            6 STORE_FAST           number
6
7 3          8 LOAD_CONST           'iterating over the list:'
8            9 PRINT_ITEM
9           10 PRINT_NEWLINE
10
11 4        12 LOAD_FAST            number
12          13 PRINT_ITEM
13          14 PRINT_NEWLINE
14          15 JUMP_ABSOLUTE        to 5
15      >> 17 POP_BLOCK
16
17 5    >> 20 LOAD\CONST            'all list items processed'
18          21 PRINT_ITEM
19          22 PRINT_NEWLINE
20          23 LOAD_CONST           None
21          24 RETURN_VALUE
```

Listing 5.31: After instrumentation

```
2          1 SETUP_LOOP          to 32
           2 LOAD_FAST           aList
           3 GET_ITER
       >>  5 FOR_ITER            to 22
           6 STORE_FAST          number

3          8 LOAD_GLOBAL         logger
           9 LOAD_ATTR           doForLog
          10 LOAD_CONST          'lineNo'
          11 LOAD_CONST          True
          12 LOAD_CONST          0
          13 CALL_FUNCTION       3
          14 POP_TOP

          15 LOAD_CONST          'iterating over the list:'
          16 PRINT_ITEM
          17 PRINT_NEWLINE

4         18 LOAD_FAST           number
          19 PRINT_ITEM
          20 PRINT_NEWLINE
          21 JUMP_ABSOLUTE       to 5
      >>  22 POP_BLOCK

          23 LOAD_GLOBAL         logger
          24 LOAD_ATTR           doForLog
          25 LOAD_CONST          'lineNo'
          26 LOAD_CONST          False
          27 LOAD_CONST          0
          28 CALL_FUNCTION       3
          29 POP_TOP

5     >>  32 LOAD\CONST          'all list items processed'
          33 PRINT_ITEM
          34 PRINT_NEWLINE
          35 LOAD_CONST          None
          36 RETURN_VALUE
```

5.2.4 Database as Log

Following tables are representing the Log: `argumentvalues`, `returnvalues`, `functioncalls`, `turinglogs`.

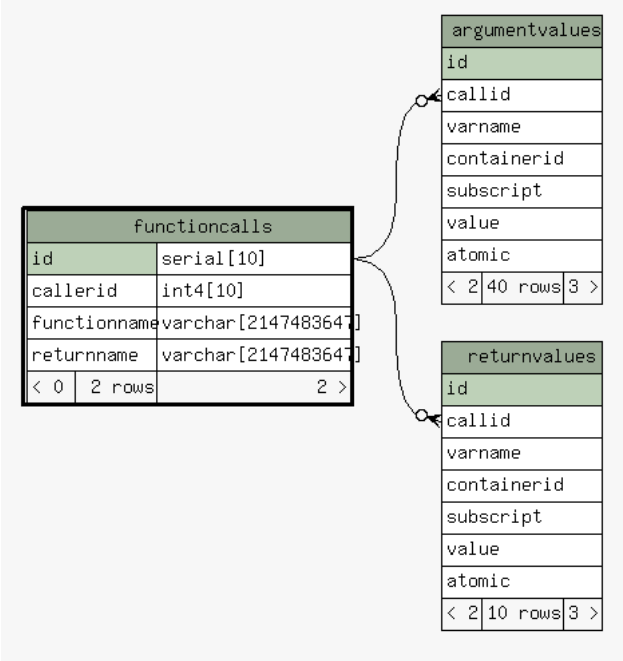


Figure 5.3: Log tables for program arguments, return values and function calls

`turinglogs` always contains same key entries: `control flow` log and `subscripts` log:

id	key	data
1	control-flow	[true, true, true, false, true , false]
2	subscripts	[0, "key1", 1, "key2", 2, 3]

Figure 5.4: `turinglogs`: Log table for control flow and collection access data

`argumentvalues` stores data logged by the `logger.doInputLog(...)` method. In the following example a list is passed as argument to the function.

Listing 5.32: Collection input log

```
1 def f(aList):
2   ...
3
4 l = ["c", "s", "d"]
5 f(l)
```

The following figure shows how the input list `l` is represented in the `argumentvalues`. The `containerid` column reference the collection, which single `atomic` entries belong to. Whereas the `subscript` says at which index the `atomic` entry is stored in the collection.

id	callid	varname	containerid	subscript	value	atomic
1	2	aList		None		f
2	2		1	0	"c"	t
3	2		1	1	"s"	t
4	2		1	2	"d"	t

Figure 5.5: `argumentvalues`: Log table for program arguments

5.3 Dependency Analysis

The analyzer component is responsible for computing AA, AR, RA and RR dependencies for all of the (called) functions in the input module. Its class structure is illustrated in the UML Diagram 5.6.

DependencyAnalysis module fetches the ctrlFlowLog, the subscrLog and the calledFunctionsLog from the database with the help of the Log module and passes the information to the ProvenanceInterpreter.

ProvenanceInterpreter class is the main component of the analyzer, since it is doing the actual work, computation of dependencies. It implements the symbolic analysis, as described in the Section 3.3. Together with the DependencyStack class and OpcodeHandlings module it builds up a custom Stack Machine, specific in the way that it is working on dependencies. The dependencies are represented with the help of the ProvenanceObject class.

The ProvenanceInterpreter, its stack - DependencyStack, and another important entity - ProvenanceObject, shall be discussed in more detail in the next section.

Writing the tool output to the database is done by the ProvenanceWriter class. It stores computed dependence provenance into appropriate tables, converting path-based variables identification into table-id-based one.

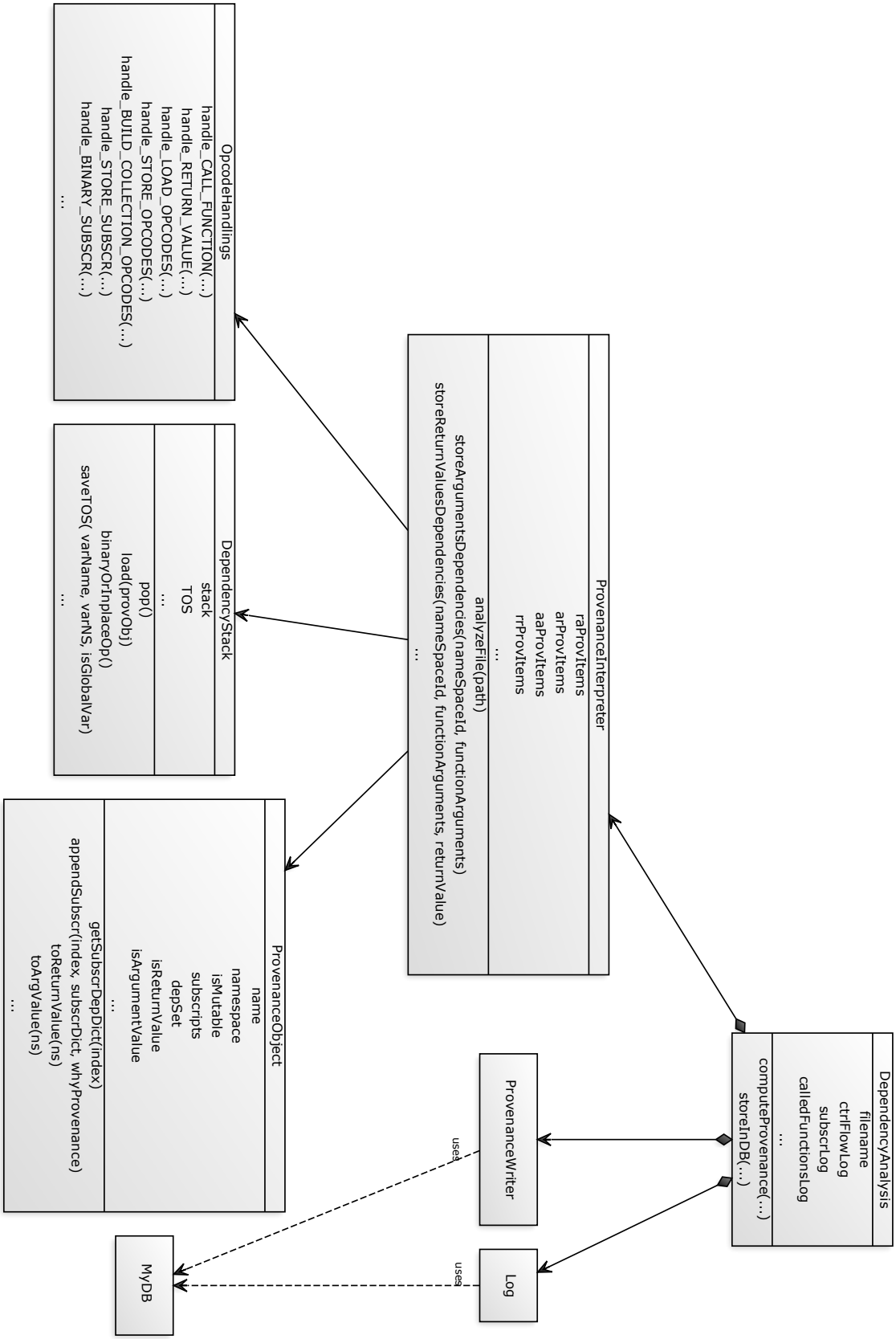


Figure 5.6: Class and module structure of the analyzer component.

5.3.1 Interpreter Concept

Instrumenter and analyzer components are similar in their basic scaffolding. In fact, both are represented through a big switch statement processing a linear stream of bytecode instructions. This is actually a common interpreter structure.

Interpretation of a program is always a simple cycle of following steps:

1. Fetch instruction, i.e opcode and opargs, PC (program counter) is pointing to
2. Increase PC
3. Execute instruction, which can effect the stack or the data or the control flow
4. Go to step 1. again

This cycle is known as the *interpretation loop*. The ProvenanceInterpreter class implements this loop.

The instruction execution in the third step of the loop is the complex part. Here, we have to define and implement a handling for each opcode, depending on what opcode's effect on the stack and on the control flow is. The implemented handlings can be found in the OpcodeHandlings module.

For the matter of instruction execution, following is worth pointing out at this point. Recall that we analyze the program *statically*, thus, no real execution in the Python virtual machine is taking place here. Instead, we *simulate* the execution by interpreting bytecode instructions *in terms of their dependence semantics*.

The interpretation implemented in the ProvenanceInterpreter class is very similar to the execution of Python programs explained in the Section 4.2. Just in the same manner, the given program is executed instruction by instruction, data is pushed to and popped from the stack, program counter is updated according to the control flow. The difference is that the "stack" and the "data" we are working with are specific. In Python, the stack is managing values bound to variable names or created by some computation. The analyzer's stack - implemented in DependencyStack class - is working on provenance associated with these values, represented by the ProvenanceObject class. This is what is meant by interpreting bytecode instructions in terms of their dependence semantics.

For instance, whenever a value would be loaded onto Python stack, this value's dependence provenance is loaded onto DependencyStack. Aggregating values on Python stack means aggregating provenance on DependencyStack.

Naturally, a big part of ProvenanceInterpreter's functionality obtains general tasks of an interpreter such as looking up variables and functions names (i.e. *name resolution*), setting the pc according to the control flow, managing data when switching between namespaces and other tasks required when executing

programs.

In following sections we take a closer look at which opcodes handlings are important for provenance computation, how exactly provenance of a single value is implemented and which part of computing provenance DependencyStack undertakes.

5.3.2 Provenance Representation

Provenance Objects

Whenever a Python object is created, a new ProvenanceObject instance associated with it is created too. It represents object's provenance and some additional provenance relevant information.

When an object is bound to a variable or is referenced as an item by a collection object, ProvenanceObject's name attribute is encoding the name of the variable or the path to the collection's item respectively.

Since a variable always points to one object, it has always one ProvenanceObject associated with it. Now, we could think, that, as several variables can be bound to the same Python object [5], the same ProvenanceObject would be associated with all these variables in this case. That is not true. We create a new ProvenanceObject for each new variable binding, i.e. ProvenanceObject class is rather describing one particular binding, than a particular object instance or a single variable.

The depSet attribute of the class implements the dependence provenance set as explained in Section 2.3.1, with the difference that references to another ProvenanceObject instances are kept, instead of saving paths (but we still can always get the correspondent path by reading ProvenanceObject's name value).

So, when one Python object is assigned to several variables, as mentioned above, a new ProvenanceObject instance is created for each binding, but they all are referencing the same depSet instance.

The same applies to objects being an item in different collections.

To illustrate this a simple example is shown in Listing 5.33.

Listing 5.33: Representing dependence provenance of an object assigned to several variables

```
1 l1 = [3,7,9]     # depSet instance d1
2 l2 = [5,11]      # depSet instance d2
3 l3 = l1
4 dict_1d = {"a": l1, "b": l2}
5
6 # ProvenanceObject (po) instances:
7 #
```

[5] Only mutable objects can be bound to different variables.

```
8 #   po l1:              po l2:              po l3:
9 # | name = 'l1' |     | name = 'l2' |     | name = 'l3' |
10 # | depSet = d1 |     | depSet = d2 |     | depSet = d1 |
11
12 #   po d["a"]:                    po d["b"]:
13 #  path: ['d', 'a']             path: ['d', 'b']
14 # | name = 'a_at_d' |           | name = 'b_at_d' |
15 # | depSet = d1     |           | depSet = d2     |
```

Mutable and Immutable Objects

In Python there are mutable and immutable objects, see [Fou13a]. Primitive types, strings and tuples are immutable. Lists, dictionaries and most other collection types are mutable. Here, an example:

Listing 5.34: Mutable an immutable objects

```
1 # immutable:
2 a = 5   # one '5' object in the heap, one reference
3 b = a   # '5' is copied, now two '5' objects in the heap
4         # and two different references for two variables
5
6 # mutable:
7 l = [3, 4]  # one list instance in the heap, one reference
8 m = l       # reference to the list is copied,
9             # still same instance of the list in the heap
10            # but now two references pointing to the same
11      # list object
```

When immutable objects are re-bound (as in lines 1 - 4), the whole provenance information is simply copied to a new ProvenanceObject, whereas ProvenanceObject instances associated with bindings to the same mutable object (as in lines 6 - 10) share the same dependency set (as we saw in the Listing 5.33).

Collections Handling

In a ProvenanceObject instance representing a collection (tuple, list, dictionary), the subscripts property of the ProvenanceObject class is a dictionary containing references to ProvenanceObject's associated with items of the collection. For instance, with `aList[0]` in Python we get the first item of the list, and with `provObjectOfaList.subscripts['0']` we get the ProvenanceObject instance associated with it.

The subscripts dictionary is of SubscriptItemsDictionary type, which implements an observable Python `dict` (see UML in Figure 5.7) and, at the same time, can observe other SubscriptItemsDictionary instances itself (see UML in Figure 5.8).

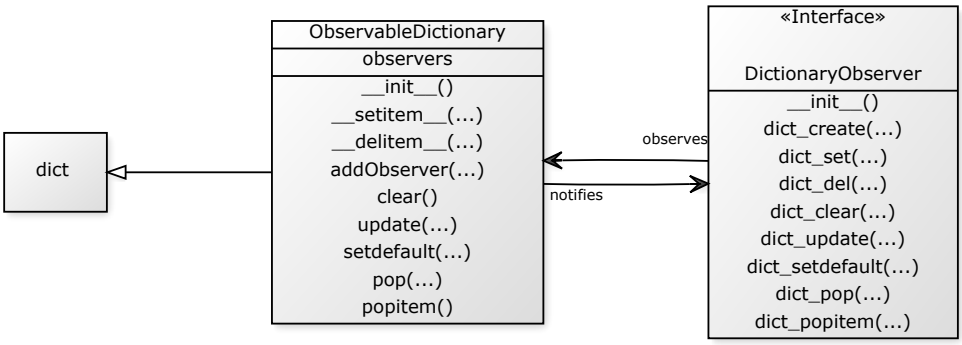


Figure 5.7: Observer pattern applied to Python's `dict`

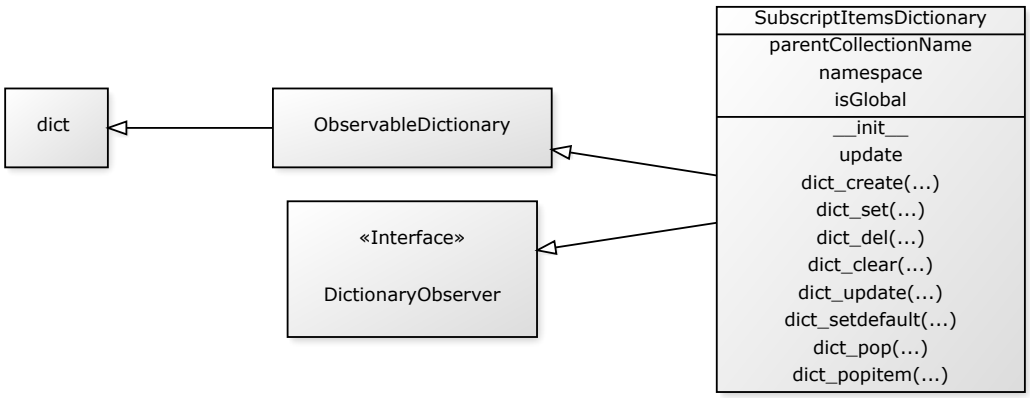


Figure 5.8: SubscriptItemsDictionary's class diagram.

Observing properties are needed for mutable collection types, i.e lists and dictionaries. Here, again, the case of multiple bindings to the same collection is interesting. As an example, look at Listing 5.35. We see that `dict_1` and `dict_2` have same subscript items but are represented by different ProvenanceObject instances. These instances must *represent* the same subscripts list but can not share the same subscripts object, as it is done with depSet, because `dict_1["a"]` and `dict_2["a"]` are *not the same bindings* and so are represented by different ProvenanceObjects too.

Listing 5.35: Subscript items

```
dict_1 = {"a": 1, "b": 100}
dict_2 = dict_1

#    po dict_1:                po dict_2:
#  path:  ['dict_1']        path: ['dict_2']
# | name = 'dict_1' |      | name = 'dict_2' |
# | depSet = deps   |      | depSet = deps   |
# | subscripts = s1 |      | subscripts = s2 |
#
```

```
10 # where
11 # -> s1["a"] references          s1["b"] references
12 #     po dict_1["a"]:              po dict_1["b"]:
13 #    ['dict_1', 'a']              ['dict_1', 'b']
14 # | name = 'a_at_dict_1' |    | name = 'b_at_dict_1' |
15 # | depSet = deps_a      |    | depSet = deps_b      |
16 #
17 #
18 #  and
19 # -> s2["a"] references            s2["b"] references
20 #    po dict_2["a"]:                po dict_2["b"]:
21 #    ['dict_2', 'a']                ['dict_2', 'b']
22 # | name = 'a_at_dict_2' |     | name = 'b_at_dict_2' |
23 # | depSet = deps_a      |     | depSet = deps_b      |
```

In case of tuples, subscripts is just providing `dict` functionality for looking up provenance of subscript items.

Marking Argument and Return Values

ProvenanceObjects associated with argument or return variables are marked as such by the isReturnValue and isArgumentValue flags. When such a ProvenanceObject occurs in the depSet of another ProvenanceObject, we know that one of AA, AR, RA or RR dependencies exists between the two variables, depending on which flag is set. This dependency can then be forwarded to the output.

It is worth mentioning here, that to differentiate to which function a ProvenanceObject is belonging to, namespace information is also kept in each ProvenanceObject instance.

5.3.3 Dependency Stack

DependencyStack class is the Python Stack's equivalent in our interpreter model. We illustrate this on a simplistic example in Figure 5.9. It shows what happens on DependencyStack, when interpreting a bytecode instruction that load a value of a variable onto Python Stack.

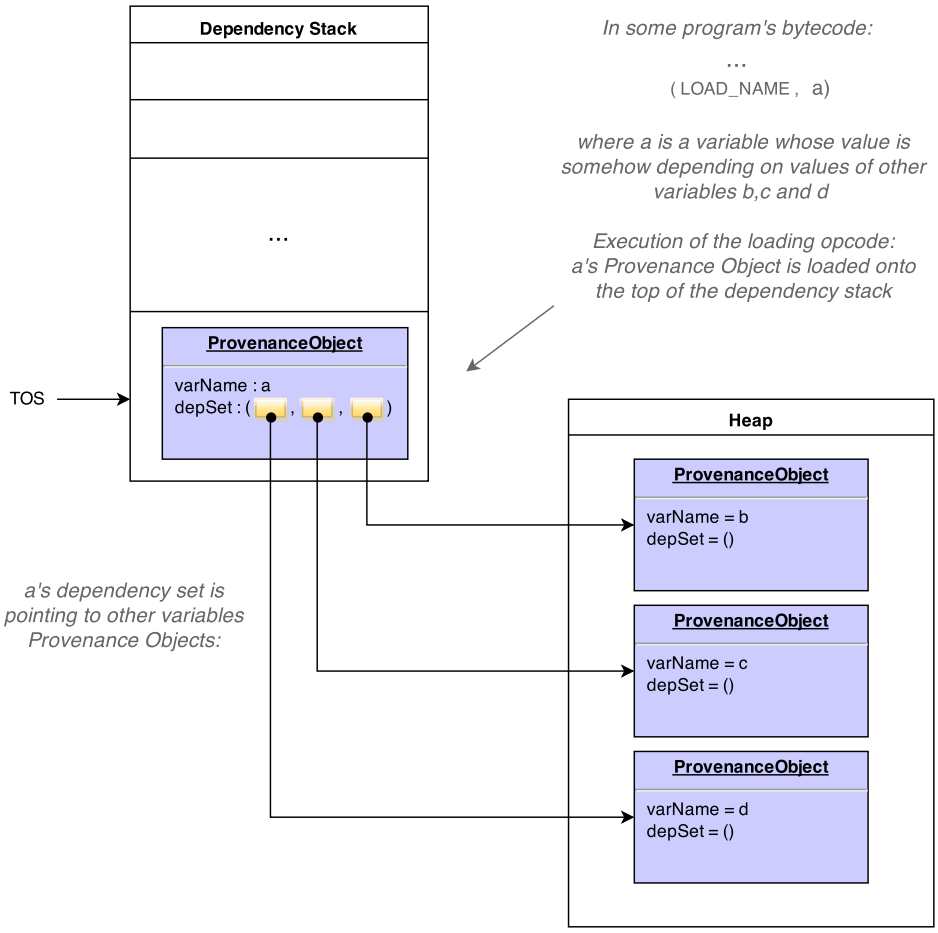


Figure 5.9: Dependency Stack: loading a variable

DependencyStack also provides kind of "dependency arithmetic", thus aggregation and propagation of dependence provenance when doing computations. Consider the program in Listing 5.36. The preprocessing in lines 3 - 5 is only there for `a` and `b` to have some dependencies for the better demonstration.

Listing 5.36: Sample program

```
1 def func(x, y, boolean)
2   # some preprocessing:
3   if boolean:
4     a = x + 1 # now a's dependency set is {boolean, x}
5     b = y + 2 # now b's dependency set is {boolean, y}
6   # computation:
7   res = a + b
8   return res
```

The computation step we want to demonstrate is the addition of `a` and `b` and returning the result in lines 6 - 8. In the graph below, the executed bytecode is representing this two lines. The figure shows what is going on on the Python Stack and on the DependencyStack, respectively, when executing these bytecode instructions.

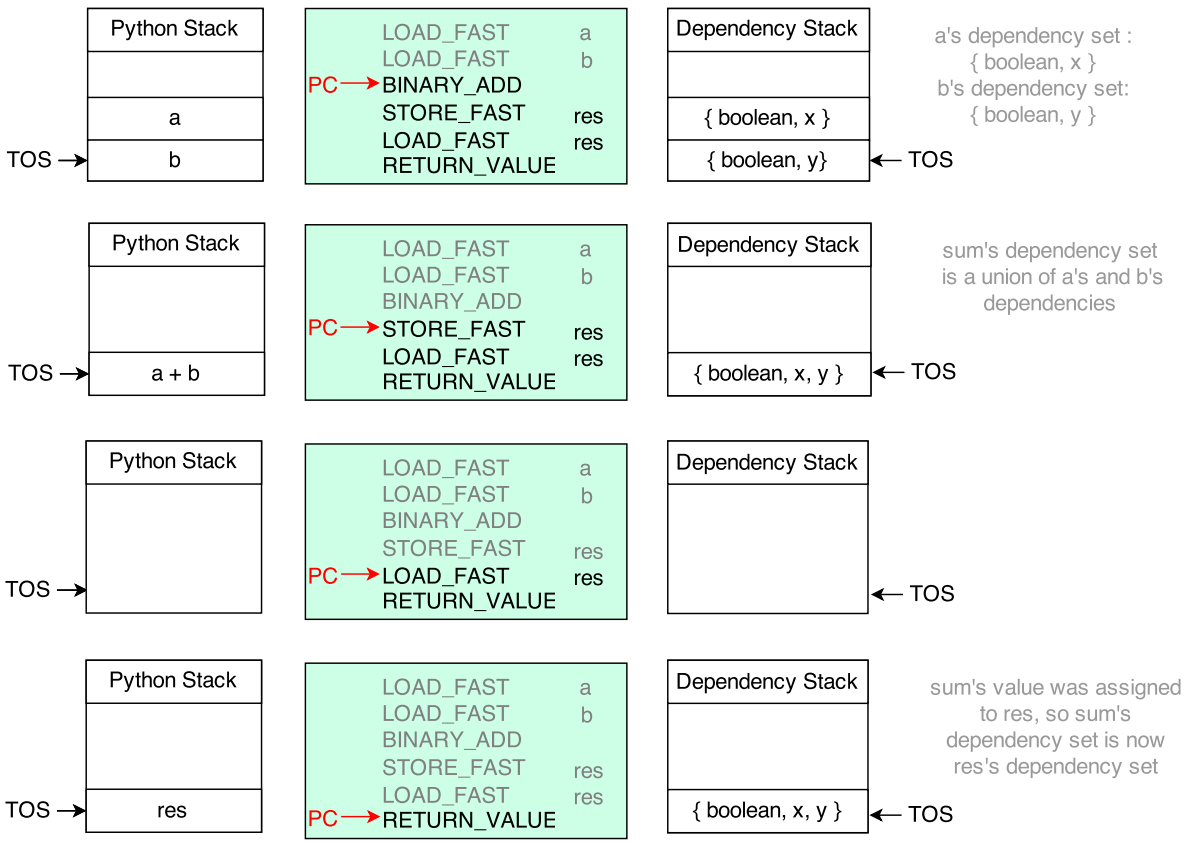


Figure 5.10: Dependency Stack: computing provenance

5.3.4 Handling Opcodes

Due to the large number of different opcodes a detailed report would be too long to be presented in this thesis, so just important code constructs and their general handling principles are given here.

In general, following must be implemented.

1. Keep track of data flow dependencies.

   All opcodes manipulating Python Stack must be handled, so that a correspondent behaviour of the DependencyStack is provided, which implies appropriate data and collections handlings, function arguments assignment, names resolving and some other issues. Technically, derivation of data flow dependencies is provided by implementing this part.

2. Keep track of control flow dependencies.

   All control flow influencing opcodes must be handled in a way that the same execution flow is taken as when executing the program in the Python interpreter. This includes jumping, switching namespaces, nested calls and recursion, and other issues. When this behaviour is achieved, tracking of control flow dependencies is provided.

Correspondent opcode handlings are implemented in respective `handle_opcode` methods in the OpcodeHandlings module.

5.3.5 Filtering Out Argument and Return Values Dependencies

ProvenanceInterpreter is executing the input program bytecode instruction for instruction, loading and moving ProvenanceObjects on the DependencyStack and creating and aggregating dependency sets of ProvenanceObjects when doing some computations.

Though all of the variables are participating in propagating of dependency information, at the end we are interested only in dependencies between function arguments and function's return values. It stands to reason that to get AA and AR dependencies, we have to look at arguments depSet values, and to get RA and RR dependencies at the depSet of the return values.

In pseudocode:

Listing 5.37: Filtering Out Argument and Return Values Dependencies

```
1 def func(a, b)
2   # at runtime, when the function is called,
3   # right here, all argument dependencies (AR, AA)
4   # are already known,
5   # write them out ->
6   ProvenanceInterpreter
7     .storeArgumentsDependencies('func',    # function name
8                 [provObj_a, provObj_b])    # arguments
9
10    ...
11
12  # just before the function returns the output to the caller,
13  # right here, all return value dependencies (RA,RR)
14  # are already known,
15  # write them out ->
16  ProvenanceInterpreter
17    .storeReturnValuesDependencies('func',  # function name
18                  [provObj_a, provObj_b],   # arguments
19                    provObj__res)           # return value
20  return res
```

5.3.6 Storing Dependencies in DB

Due to efficiency issues when reading computed provenance from the database, dependencies are stored in two tables `argumentvaluesrelations` and `returnvaluesrelations`, and are to be seen from two perspectives respectively: dependencies of argument values and dependencies of return values. For that, the same dependence provenance set is mirrored. E.g., for a dependency `(return1, arg1)`, `{return1}` is stored as `forward_ret` in the `argumentvaluesrelations` table, and `{arg1}` is stored as `backward_arg` in the `returnvaluesrelations` table.

The `id` column in the two tables is referencing argument `id` in the `argumentvalues` or return value `id` in the `returnvalues` tables respectively.

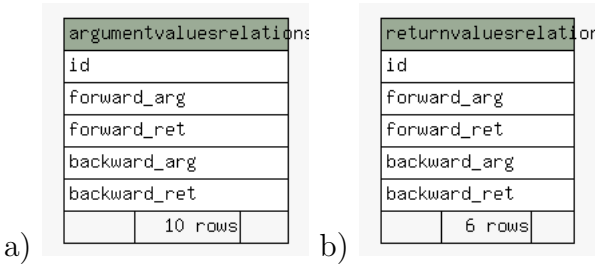


Figure 5.11: `argumentvaluesrelations` and `returnvaluesrelations`: tables storing dependencies

5.4 Notes

5.4.1 Input Restrictions

It is assumed that input programs are written in functional programming style, i.e. they always have a variable returning statement (so simple `return` or `return None` is not allowed).

In the for-loops we assume iterations over collections.

We concentrate on function and module code objects only, and do not regard classes. Thereby, only Python's *user defined functions* are analyzed, see "Callable types" in [Fou13a]. Object methods are assumed not to be used.

5.4.2 Excluded source code structures

Following code constructs and expressions are either not implemented or were not tested:

1. inline declarations, e.g. `return ((x + y)/z )`
2. nested function definitions:

   Listing 5.38: Nested function definitions are not allowed

   ```
   def f:
     ...
    def g:
      ...
     return x
    ...
   return y
   ```

3. `try ... catch ...`
4. `for ... else ...` , `while ... else ...`
5. `object.attribute` or `object.method()`
6. usage of global variables:

   Listing 5.39: `global` statement not allowed

   ```
   global varName
   varName = 3
   ```

7. list comprehensions
8. variable args: `def f (a,b, *args, **kwargs)`
9. `break` and `continue` statements
10. handling of the most built-in functions is not implemented (though an extendible interface is provided)

5.4.3 CPython Bytecode

All code in this thesis was written and tested against Python 2.7.5. Other (compiler) versions could translate same source code statements and constructs to different bytecode representations and structures, which may lead to wrong interpretation by this tool and thus to wrong provenance results or even crashes while running. For the list of opcodes our interpreter is working on see Python 2.7.5 Documentation of the `dis` module.

6 Conclusion and Outlook

6.1 Conclusion

This thesis concentrated on the implementation of a tool for instrumentation and analysis of CPython bytecode with the aim to compute data provenance in Python programs. We successfully implemented a working component and integrated it into the existing Python provenance analyzing workflow.

The provided tool computes dependencies between the input and the output of a given Python program. Input programs effective log their run time data while execution. The custom interpreter we have built executes input programs in a symbolic way and derives dependencies between arguments and return values. Bytecode handlings of different code constructs needed for that were implemented and tested. The Stack Machine approach of the implemented interpreter has proved to be a good concept.

Finally, several programs generated by a prototype system for query translation have been successfully analyzed by the tool. The result dependencies have been then successfully forwarded to the visualisation component of the workflow.

6.2 Future Work

There are several improvements that might be made on the current implementation, which we outline below.

We would like this tool to provide provenance output that differentiate between why and where dependencies. This differentiation would explain dependencies in a query in more detail. When properly visualized it could help the user to better understand the data.

Tool's correctness should be verified through more representative tests. Generally, we would like to make the current implementation more robust and harder to break by any unexpected code constructs in the input program. In particular, all the issues mentioned in Section 5.4 require more attention as could be invested in this thesis.

Tool's performance should be investigated and probably optimized. Notably, performance overhead due to the additional functionality when running the instrumented program should be minimized. The same applies to the space

overhead for storing the Log. It would be also interesting to compare performance of this tool and that of the AST implementation.

Future work is not only necessary to extend, verify and optimize some features of the tool, but also to clear and formalize some dependence and provenance interpretation issues. For instance, it would be interesting to investigate the similarity between data provenance analysis and dependence analysis as understood in compiler theory in a more formal and detailed way.

Finally, we can imagine the concept implemented here to be adapted to a lower level language, e.g. for better performance or as a common interface to queries compilation (when one would take the language used in a particular DBMS).

List of Figures

Listings

List of Abbreviations

PBIM Python Bytecode Instrumentation Module
DBMS Database Management System

References

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160. ACM, 1999.

[Bet14] Janek Bettinger. Implementation of a Browser-based Interface for Provenance Analysis, 2014. `http://db.inf.uni-tuebingen.de/theses.html`;.

[BKWC01] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In *Database Theory, ICDT 2001*, pages 316–330. Springer, 2001.

[CAA07] James Cheney, Amal Ahmed, and Umut A Acar. Provenance as dependency analysis. In *Database Programming Languages*, pages 138–152. Springer, 2007.

[CCT09] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases*, 2009.

[Che07] James Cheney. Program slicing and data provenance. *IEEE Data Eng. Bull.*, 30(4):22–28, 2007.

[CWW00] Yingwei Cui, Jennifer Widom, and Janet L Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000.

[Fou13a] Python Software Foundation. Python data model, May 2013. `https://docs.python.org/release/2.7.5/reference/datamodel.html`;.

[Fou13b] Python Software Foundation. The python language reference/ alternate implementations, May 2013. `https://docs.python.org/release/2.7.5/reference/introduction.html#alternate-implementations`;.

[Fou15] Python Software Foundation. Python bytecode, April 2015. `https://docs.python.org/2.7/glossary.html#term-bytecode`;.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift