Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Lehrstuhl für Datenbanksysteme
Wintersemester 2013/2014

Bachelorarbeit in Informatik

Instrumentation of Python Bytecode and Symbolic Program Analysis

Author:
Simon Kalt

Supervisors:
Prof. Dr. Torsten Grust
Tobias Müller

Contents

Abstract

As part of a project to help non-programmers understand computer programs, the goal of this work was to generate information on the behaviour of Python programs. The implementation described in this work uses an approach based on instrumenting Python bytecode to trace back variable assignments in order to build a graph depicting how the output of the program depends on its intermediate and input values. The presented version of the implementation enables the user to analyze simple programs by visualizing the data flow occurring during to the computation of a program's output.

1 Introduction

This work's main objective is the analysis of Python programs and finding a correlation between the input and output of these programs. The project from which this idea originated has the goal of helping people, who are not yet well versed in computer programming, understand existing or self-written programs.

The analysis works by relating values used throughout a given program to the input relevant for generating them. The result is then presented as a directed graph, visualizing the dependence of the output of the program on the input values given.

The implementation uses an approach based on inspection, manipulation and execution of the bytecode generated by the compiler of the CPython implementation of the Python programming language [3].

2 Tools

2.1 Byteplay

The implementation presented works primarily with Python bytecode. In the CPython implementation of Python, the source code of a program is compiled into an list of bytecode instructions. These instructions are simple operation codes with a single optional argument. The bytecode is then executed by an interpreter. This approach is in general much faster than interpreting the source code of a program directly.

In CPython bytecode branches and loops are represented through conditional and unconditional jumps. This means, that the code can be stored in a list, as opposed to a tree, making it easy to analyze and modify.

For easy access to the bytecode, the library *Byteplay* [5, 6] is used. Byteplay transforms Python code objects into Byteplay objects which contain bytecode in a human-readable form that is also easy to manipulate programmatically. The bytecode in such an object is kept in a list.

Each instruction in the original bytecode is translated into a tuple consisting of a constant representing the instruction and a Python value representing its argument. For instructions without arguments, the Python value `None` is used as the second component of the tuple.

The bytecode examples in this work are either represented similarly to the output of the Python disassembler (the `dis` method in the module of the same name) or as parts of pretty printed Byteplay objects.

2.1.1 Jumps and Their Targets

In Python bytecode, jump targets are represented as absolute positions in the bytecode or as offsets from the current position. To allow for arbitrary insertion and removal of instructions, the targets of jumps are resolved during the transformation and replaced by instances of the `Label` class defined in the Byteplay module. The jump-instruction then holds a reference to a specific `Label` object and another reference to the same object is inserted at the appropriate place in the list of bytecode instructions.

Consider, for example, the bytecode given in Listing 2.1, where the second instruction is skipped. This is done through a forward jump with offset 1, since the `BINARY_ADD` instruction is 1 byte long.

This code sample would result in the Byteplay representation seen in Listing 2.2, where the target of the jump is an object. Later, during the reconversion into actual bytecode, the correct distance to the target — or, for absolute jumps, its position — is

```
1 JUMP_FORWARD, 1
2 BINARY_ADD
3 BINARY_SUBTRACT
```

Listing 2.1: Representation of jumps in Python bytecode

calculated and inserted, relieving the user of worrying about jump targets while editing the bytecode.

```
1 [
2 (JUMP_FORWARD, <byteplay3.Label object at 0x1007dce10>),
3 (BINARY_ADD, None),
4 (<byteplay3.Label object at 0x1007dce10>, None),
5 (BINARY_SUBTRACT, None),
6 ]
```

Listing 2.2: Representation of jumps in a Byteplay object

2.1.2 Constants and Variables

Python code objects contain, next to the actual bytecode, several tuples with information. Two of these tuples hold constants and variables used in the code. In the code, values are referenced by their indices in these tuples. Byteplay resolves each of these references and inserts the constant or the name of the variable in its place.

Say our code loads a constant 'abc' which is stored at index 3 of the code's tuple of constants. In bytecode the instruction LOAD_CONST is called with the argument 3. After conversion into a Byteplay object, this instruction reads (LOAD_CONST, 'abc'). This allows the user to change existing constants, and introduce new ones. The concept is the same for variables, with the difference that the code's variable-tuple contains the names of the variables.

2.2 Instrumentation and Logging

In the first step, the bytecode of the input program is *instrumented*. This means that additional instructions are inserted into the bytecode, which do not alter the result of any computations or the flow of the program. These instructions do however record information during the execution of the program.

This information is stored in an instance of the Logging class which is passed to the CPython interpreter as a global variable. The Logging class provides several methods to record information on the results of jump conditions, subscript indices for array, tuple and dictionary access, values of loaded object attributes, and iteration counters of for-iterators.

For a single module, the information about jump conditions, subscripts and attributes are kept as lists. Each time a value is logged, it is simply appended to the corresponding

list. The for-iterators are stored in a dictionary. Each entry is identified by the iterator's line number and holds an integer indicating how many times the iteration was performed.

After the execution, this information is extracted from the logging object. The data is then used during the execution in the self-implemented stack machine, where it helps in deciding on the program flow using the logged information on jump conditions, object attributes and for-iterators. The data on subscript operations, along with the names of the logged object attributes, is used for naming values during the execution.

2.3 Execution in Self-implemented Stack Machine

In the second execution step, a self-implemented stack machine is used to interpret the original bytecode of the input program. This implementation works with Byteplay objects and the lists of bytecode instructions contained within them.

2.3.1 Value Representation

During the interpretation of the bytecode, all values on the stack or in the variable pool are instances of the class `Value`. An object of this class contains — among other auxiliary information — an identifier for the value, which is a combination of a name and a version, and, if available, a Python value. An instance of this class with the name `x` and the Python value `True` is here represented as `Value(name = 'x', value = True)`.

2.3.2 Dependency Tracking

As functions are executed on value objects, new instances and versions are created, which partially depend on other values in the program. This is done in a similar way to static single assignment (SSA) form. Whenever values are combined, the result is saved in a new version of the target variable.

```
1 x = 10
2 y = 11
3 y = y + 1
4 x = x + y
```

Listing 2.3: Assignment and overwriting of variables

For example, the program snippet given in Listing 2.3 would be executed as the following:

$$
\begin{aligned}
x_0 &\leftarrow 10 \\
y_0 &\leftarrow 11 \\
y_1 &\leftarrow y_0 + 1 \\
x_1 &\leftarrow x_0 + y_1
\end{aligned}
$$

First, x and y are initialized as version 0. When a new value is assigned to y, a new version of y is created. In the following addition, x is version 0 and y is version 1. Afterwards, the result variable x is in version 1 as well.

During this process, each time a new version is created, the values used for its computation and their specific versions are recorded. Through this, we can determine, that x_1 depends on the values x_0 and y_1 and y_1 depends on y_0 and the constant value 1. The dependency relation for x_1 therefore looks like Figure 2.1.

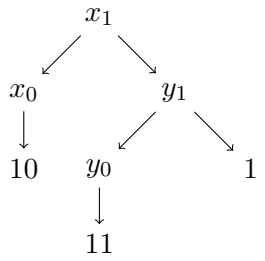


Figure 2.1: Dependency relation for x_1 with the SSA approach

2.3.3 Value Computation

During the interpretation, most of the Value objects on the stack and in the variable pool do not carry actual data values. A computation does not actually compute anything. Instead it collects information on which input values and variables were used.

While the results of internal operations such as addition usually depend on all input variables, the same is not necessarily true for user implemented functions. So wherever possible, the interpreter enters the bytecode of called functions and continues the tracking of dependencies inside. When the function returns, the return value contains detailed dependency information originating from the computations within the function.

Consider the definition of func in Listing 2.4. This function takes three arguments and returns a result, which does not depend on the third argument z.

```
1 def func(x, y, z):
2     z += 1
3     return x + y
```

Listing 2.4: Example function func

If this function is called as func(3, 4, 5) the returned Value object contains references to x and y as dependencies, which in turn depend on the constant values 3 and 4, respectively. This results in the dependency graph shown in Figure 2.2.

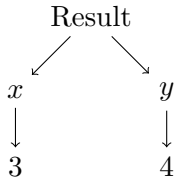


Figure 2.2: Dependency graph for the function call func(3, 4, 5), note the absence of z

2.3.4 Program Flow

Keeping almost no values throughout the execution poses a problem: when encountering instructions where the program flow is dependent on the results of previous computations, it is — in most cases — not possible to decide which action to take based on the computed (or rather not computed) values. This is where the information collected during the instrumentation step comes in: whenever a conditional jump is encountered, a boolean value is removed from the list of jump conditions and used to decide whether the jump should be performed or not.

A similar process is used with for-iterators. The collected information contains the maximum number of iterations for each for loop. When the execution is started, an additional value is added and initialized as 0, representing the number of times a specific loop has been executed. This value is then incremented with each iteration of the specific loop. When the previously recorded value is reached, the loop is no longer executed. To support nested for loops, an iterators execution counter is reset to zero each time the maximum number of iterations is reached, allowing it to be executed again.

Whenever an attribute of an object is loaded, the actual value of that attribute is taken from the list of object attributes and added to the stack.

2.3.5 Variable Pool

When variables are stored or arguments are passed to a function, the corresponding values are stored in a variable pool. In this implementation, the variable pool is a dictionary indexed by the names of the variables. For each variable name, a dictionary indexed by *levels* is kept.

The level, initially zero, is increased every time a function is entered during interpretation. This ensures, that variables defined in subroutines do not overwrite variables from higher levels while still allowing access to these values.

After the execution on a certain level is finished, all variables defined on that level are deleted, restoring the variable pool to its previous form.

2.4 The Dependency Dictionary

The dependency dictionary is the data structure used to keep information on the dependencies of different versions of used values. Each key in the dictionary represents the name of a value. In the example shown in Listing 2.4, the keys would be func, x, y and

z. The value associated with each key is a list of the different versions this named value appears in. Each time a new version of a value is created, its dependency information is added to the end of this list.

A certain version is also represented by a dictionary. Each of these dictionaries contains a set of named and a set of unnamed dependencies. Named dependencies are used for values that have a name, and through that have at least one entry in the dependency dictionary. These dependencies are recorded as tuples of their name and the version used. For example: in line 2 of the example, z is incremented by 1. This causes a new version of z to be added to the version list for z's entry in the dependency dictionary. Since the new value depends on the old value of z, the named-set of the new version would contain an entry ('z', 0) (considering the version of z used was version 0).

Unnamed dependencies are Value objects of values without names — and therefore without version information — used in the computation of a value. To continue the example: incrementing z by 1 makes the new version of z not only dependent on the used version of itself but also on the number 1. Since this number is used directly and not as a named variable, it has neither name nor version information. Therefore the value, which should be something similar to Value(value = 1), is added to the unnamed dependencies of the new version of z. The entry in the dependency dictionary for z therefore contains the information shown in Listing 2.5.

```
1 'z': [
2         {'named': set(),
3          'unnamed': {Value(value = 5)}}, # version 0
4         {'named': {('z', 0)},
5          'unnamed': {Value(value = 1)}}  # version 1
6     ]
```

Listing 2.5: Dependency dictionary entry for z, set() denotes an empty set

As we can see, the first version of z solely depends on the constant value 5 passed to the function. When the result of $z + 1$ is stored under the name z the second entry is created. This version depends on, as described before, the used version of z and the constant value 1.

3 Steps of Execution

The analysis of a program consists of multiple steps. These steps are illustrated in Figure 3.1.

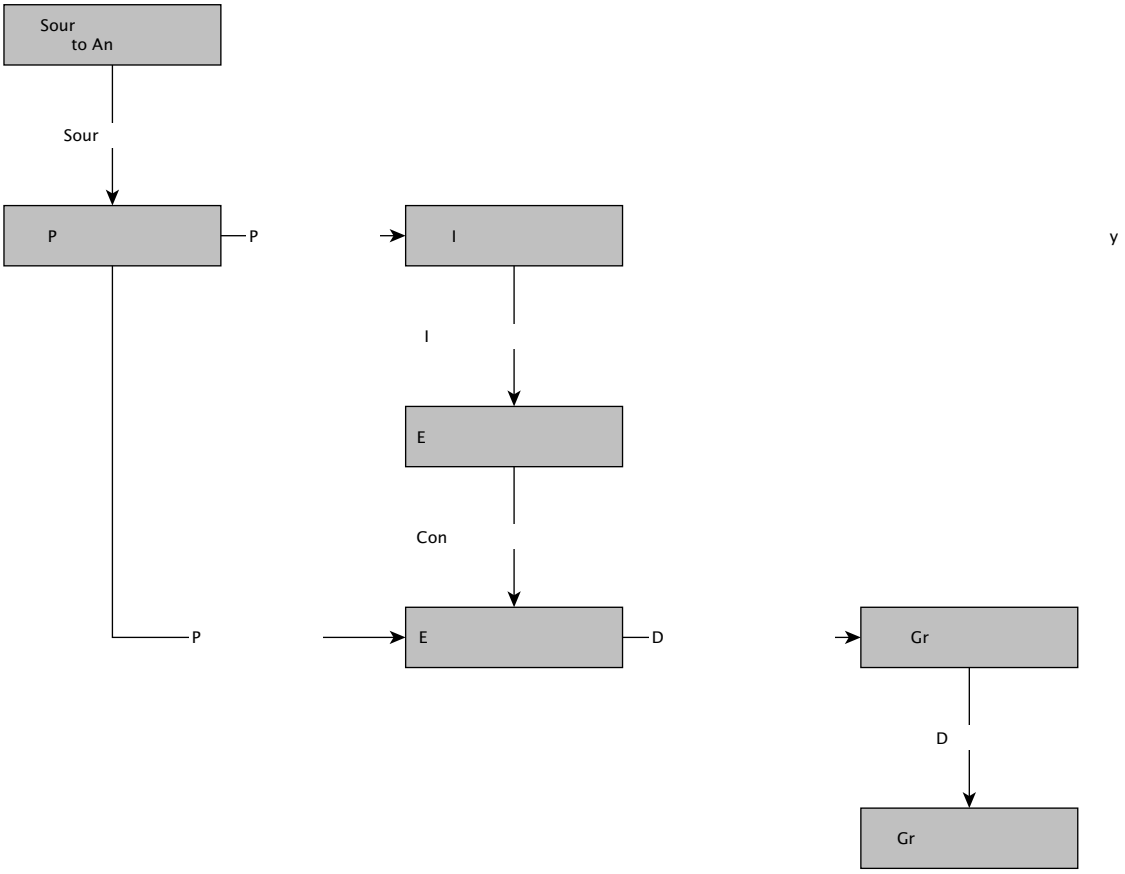


Figure 3.1: Program Flow

First, the source code is read from the given program. This source code is then compiled to Python bytecode. In a step called instrumentation, this bytecode is now extended to record certain data during execution. This includes information on the program's control flow, indices used to access elements in collections, attributes loaded from objects, and counters of iterations.

This instrumented code is then executed using the virtual machine of the CPython implementation. Alongside we pass an instance of the Logging class, which contains the necessary functions, which perform the actual logging, referenced in the instrumented bytecode. After the execution, this object contains the recorded information.

This information is then used during the execution of the program in a self-implemented stack machine. This results in a data structure — the dependency dictionary — where a value used in the program is identified by a name and a version and contains references to all values influential to its computation.

In the third step, a graph for the return value of the program is calculated from the dependency dictionary. This is done by connecting the value to every value recorded in its dependencies. This process is then recursively used for all of these dependencies, resulting in a graph visualizing the relation of the result value to the input parameters and intermediate values of the program.

3.1 Instrumentation

The first step in our program analysis is the instrumentation. In this step the source file is read into a string, which is then compiled to Python bytecode using Python's internal compiler. This compiled code is then transformed into a Byteplay object, which is then extended in various ways.

We iterate over the list of bytecode instructions. In four cases, we take special action by inserting additional bytecode instructions. These events are: conditional jumps, subscript operations, loading of attributes, and iterations which use Pythons built-in iterators.

This step is important, since the self-implemented stack machine does not calculate actual values. This means, that without the information collected during the execution of the bytecode generated in this step, it would not be possible to decide the control flow of the program.

After the instrumentation, we use the marshal module built into Python to serialize the bytecode and save it to a file.

In the following examples, we assume we are working with a file called example.py.

3.1.1 Conditional Jumps

Conditional jumps are used to branch the program. They are used to implement conditional statements and while loops. The conditional jump instructions expect the jump condition to be on top of the stack. A typical use of a conditional jump can be seen in Listing 3.1. Before the jump instruction is evaluated and the jump is made (or not), we insert the bytecode shown in lines 4–10 of Listing 3.2.

This code first duplicates the jump condition (4), then loads the name of the module we are working on (5) and the log_boolean function of the logging object (6–7). Then the function is moved on the stack (ROT_THREE), since the function call in line 9 expects the arguments before the actual function. The unused return value of this function call

```
1 LOAD_FAST, x
2 LOAD_FAST, y
3 COMPARE_OP, ==
4 POP_JUMP_IF_FALSE, <byteplay3.Label object at 0x1007bcb50>
```

Listing 3.1: Example of a conditional jump

```
1 LOAD_FAST, x
2 LOAD_FAST, y
3 COMPARE_OP, ==
4 DUP_TOP
5 LOAD_CONST, 'example'
6 LOAD_GLOBAL, 'logging'
7 LOAD_ATTR, 'log_boolean'
8 ROT_THREE
9 CALL_FUNCTION, 2
10 POP_TOP
11 POP_JUMP_IF_FALSE, <byteplay3.Label object at 0x1007bcb50>
```

Listing 3.2: Instrumented version of the conditional jump in Listing 3.1

is removed by POP_TOP, leaving the stack in the state it was in, after line 3 had been executed.

Through insertion of this bytecode we ensure, that each time a conditional jump instruction is evaluated during the execution of the instrumented bytecode, the jump decision is added to the list in the logging object.

3.1.2 Subscript Operations

Subscripts are used as indices to access elements in lists, tuples and dictionaries. In lists and dictionaries they are also used to store and delete them. For these operations, we record the value of the index accessed.

For example: we have the instruction shown in Listing 3.3, where the index stored in the variable `i` in the list `xs` is accessed. The bytecode produced by the Python compiler is shown in Listing 3.4. It works by first adding the contents of the variables `xs` and `i` to the stack and then calling the BINARY_SUBSCR operation on these values. The instrumented code, inserted before the BINARY_SUBSCR operation, can be seen in lines 3–9 of Listing 3.5. Essentially, we record the index of the subscript operation in the same way as we record the boolean condition of a conditional jump.

```
1 x = xs[i]
```

Listing 3.3: Example of a subscript operation

After the first execution, the list of subscripts contains all subscripts used during the execution of the program in the exact order they were used in. This is then used in

```
1 LOAD_FAST, xs
2 LOAD_FAST, i
3 BINARY_SUBSCR
4 STORE_FAST, x
```

Listing 3.4: Bytecode for the subscript operation in Listing 3.3

```
1 LOAD_FAST, xs
2 LOAD_FAST, i
3 DUP_TOP
4 LOAD_CONST, 'example'
5 LOAD_GLOBAL, 'logging'
6 LOAD_ATTR, 'log_subscript'
7 ROT_THREE, None
8 CALL_FUNCTION, 2
9 POP_TOP
10 BINARY_SUBSCR
11 STORE_FAST, x
```

Listing 3.5: Instrumented bytecode for the subscript operation in Listing 3.3

the execution in our custom virtual machine to relate the values that are read from or stored in collections, not just to their parent object, but also to the index accessed. This is done by adding the index to the name of the retrieved value.

3.1.3 Attribute Loading

Since there is no value computation, attributes of objects can often not be found during the execution in the self-implemented stack machine. Because of this we record the attributes and their names in a list. This is, again, done in a very similar way to the recording of the jump conditions and subscripts. Here, however, we record the value on top of the stack *after* the LOAD_ATTR instruction has been executed. After this instruction, the value on top of the stack is the wanted attribute, which is then recorded in the list of the logging object.

This list of attributes then allows us to use the value and name of the attribute during the execution in our custom virtual machine. If the value is a function that is called, it allows us to track the execution inside this function.

3.1.4 Iterations

For-iterators are used in Python's for loops and list comprehensions. They work differently from while-loops as an iterator of the given iterable object is used to subsequently assign each of that object's elements to the variable given in the code. For example **for** x **in** xs assigns the next value of xs to x in each iteration until there are no more elements left. Because of this approach, no list-indices are kept during runtime. In the bytecode the instruction which decides on and executes the iteration is called FOR_ITER.

The iterators are logged in two steps. Before the FOR_ITER instruction we insert code which clears the value of the iterator before it is executed. The inserted code is displayed in Listing 3.6. Through this the indices of lists in nested loops are recorded correctly later on. The inserted code works by loading the line number of the iterator in the original bytecode (1), loading the module name and iterator-clearing function and calling it on the arguments in the correct order (2–7) as seen before.

```
1 LOAD_CONST, <line in original bytecode>
2 LOAD_CONST, <module name>
3 LOAD_GLOBAL, 'logging'
4 LOAD_ATTR, 'clear_for_iter'
5 ROT_THREE
6 CALL_FUNCTION, 2
7 POP_TOP
```

Listing 3.6: Instrumentation of for-iterators

After the FOR_ITER instruction we insert code which — if the loop is entered — updates the iterator's number of iterations. The update function also records the current index, which technically doesn't exist during execution, in the list of subscripts for naming the accessed value later on. This code is the same as the code inserted before the iteration instruction, except that now the function update_for_iter is loaded and called instead of clear_for_iter .

3.2 Execution in CPython VM

In this step, the instrumented bytecode from the previous step is executed by the CPython virtual machine to perform the logging of information with the previously inserted bytecode instructions.

For this, the bytecode file written after the instrumentation, is loaded using the loads method of the marshal module. Then we create an instance of the Logging class, which contains the functions referenced in the instrumented bytecode. Now the instrumented code is executed using the CPython interpreter (**exec** function) with the logging object passed along as a global variable. After the execution, the relevant information is extracted from the logging object.

3.3 Execution in Self-implemented Interpreter

The original source file is now read and compiled again, in the same way as in the instrumentation phase. Then the self-implemented interpreter is called.

This interpreter is a partial re-implementation of the CPython virtual machine. It consists of a loop iterating over a set of bytecode instructions. For each instruction certain actions are taken. These actions mostly pertain to relating generated values to the values used by the instruction. An important concept for the implementation of this

virtual machine was, that as few operations on the input data as possible were actually performed. This allows us to easily retain multiple versions of data of arbitrary size, since there is no need for a version of a data value to contain the actual Python value.

In the interpreter, the code is first transformed into a Byteplay object. If there are arguments present, which happens when the interpreter calls a function, they are entered into the variable pool at the current level. The bytecode is then interpreted one instruction at a time.

During the execution, values on the stack and in the variable buffer are assigned versions. Each time a value changes its version changes as well. Each version of a value or variable used in the program being analyzed holds a set of dependencies. These dependencies reference other values in certain versions, enabling the program to accurately relate the result of a computation to its input.

The handling of the bytecode instructions in the interpreter is modeled after the descriptions given in the Python documentation [4]. Here, however, (almost) no values are calculated. When, for example, a multiplication instruction is executed, the resulting value only references the input values in their respective versions. No actual multiplication is performed.

3.3.1 Stack Manipulation

Stack manipulating instruction work as they are described in the documentation [4]. For example ROT_TWO switches the first value on the stack with the second one and DUP_TOP duplicates the value on top of the stack. An exception to this is POP_TOP: here it is checked, whether the instruction is the third to last in the bytecode and if the last three instructions form the sequence shown in Listing 3.7.

```
1 POP_TOP
2 LOAD_CONST, None
3 RETURN_VALUE
```

Listing 3.7: Last three instructions in a Python module

This is the case at the end of any Python module and thereby source file, and also at the end of a function with no return value. If this is the case, the value that was just popped from the stack is returned to the caller. For functions with no return value, this doesn't matter, since the result (which would usually be None) is popped off the stack by the caller. For the module, this returns the result of the last expression as the result of the entire module. In all other cases, POP_TOP works just as expected, removing the value on top of the stack.

3.3.2 Binary and Unary Operations

Built in binary, in-place and comparison operations such as BINARY_ADD, INPLACE_ADD or BINARY_MULTIPLY are all executed in the same way, as they all use exactly two arguments and produce a result depending on both of these arguments. For the result,

a new Value object is created containing references to the two arguments, which is then added to the stack.

Unary operations create a new version of the used value — or, if that value does not have a name, a new Value object – which depends on the current version of the value. This updated value is then pushed back onto the stack.

3.3.3 Jumps

Conditional jumps make use of the list of conditions collected in the instrumentation step. The topmost value of this list is used to determine, whether the jump is made or not. A new Value object depending on the calculated jump condition is created with a name containing information about the jump and an expiration argument pointing to the jump-target.

If the jump is not made, this object is then added to the set of block dependencies. During execution, the elements in this set are added to the dependencies of all calculated values. The jump condition remains in this set until the target of the conditional jump is reached in the bytecode. Through this, all code between the jump instruction and its target, if it is executed, depends on the jump condition.

The actual jump is performed by iterating through the bytecode until the jump target is reached. This is also how unconditional jumps are made.

3.3.4 Collections

Subscript operations use the list of subscripts. When accessing a value from a collection, a new value referencing the collection and the index is added to the stack. The new value is given the name of the collection plus the value of the logged subscript in brackets, resulting in value names such as xs[3] or dict[key].

Deletion and storage is handled by adding the value used as the index and — in case of storage — the value to be stored, to the dependencies of a new version of the collection. Since these bytecode instructions remove the collection from the stack, it is only possible to track these operations on named Value objects.

This is resolved by naming list, tuples, sets and dictionaries upon their creation: if all elements added to the collection have names, they are listed in the name, surrounded by parentheses matching the type of the collection. For example a list built from the values x, y and z would receive the name [x, y, z].

If not all values added upon creation have names, the collection is named with the created Value object's unique Python ID, resulting in names such as List<0x1018eb3f8> or Tuple<0x101891d88>. Dictionaries (internally called maps) are always created without content, therefore receiving names such as Map<0x10183b190>.

3.3.5 Functions

The creation of functions follows the Python documentation for the MAKE_FUNCTION instruction: arguments, keyword-arguments and their default values, the function code, and the function's qualified name are removed from the stack and used to create an

instance of the Function class. The resulting object is then wrapped in a Value object and added to the stack. The dependencies for the function are set to all arguments used in its creation.

When a function is called using the CALL_FUNCTION instruction, the interpreter examines the function object. If it is a built-in function, meaning it has no bytecode, a string representing the function call is built, this string looks like a function call in Python code: the name of the function, followed by the list of arguments in parentheses. If the function is an instance of the previously mentioned Function class, the objects execute_function method is called. This function just calls the interpreter with the functions code after setting the default parameters, if necessary. After execution, the return value of the function is returned alongside the updated dependency dictionary.

3.3.6 Object Attributes

Loading of attributes, such as in the Python code x = obj.y, is handled by retrieving the actual value of the attribute from the list of attributes, created during the execution of the instrumented bytecode. To the created value object, which receives a name such as parent.attribute, information on the attribute's parent is added.

Storing attributes works by combining the names of parent object and attribute, and adding an entry to the dependency dictionary referencing the parent object and the logged value of the attribute.

Deleting attributes works by simply setting the object and the attribute name as the dependencies of a new version of the object.

3.3.7 Variables and Constants

When loading variables, the variable pool is first checked for the name to be loaded. If the name exists, the pool is searched upwards starting at the current level, until a value is found. If this is not the case, and the loading-instruction was LOAD_GLOBAL, the global variables are searched for a value of the given name. Finally, the builtin module, which contains functions built into Python, is searched.

If a value is found, it is, if necessary, wrapped in a Value object, and added to the stack. Constants are loaded by simply wrapping the argument to LOAD_CONST — which, in Byteplay, is the constant — into a Value object and adding it to the stack.

Storing variables works by adding an entry to the dependency dictionary with the name the value should be stored under. If the value is to be stored globally (STORE_GLOBAL), it is added to the global variables. Otherwise it is added to the variable pool under the given name and the current level.

3.3.8 Loops

In CPython, loops have certain utility bytecode instructions. Before the start of a loop, SETUP_LOOP is called. The argument for this instruction is a Byteplay label pointing to the end of the loop. We record this information by adding the label and the current size of the stack to a list of block sizes.

The teardown of loops is done by the instruction POP_BLOCK which typically occurs directly before the label SETUP_LOOP is pointing to. Its job is to remove all leftover information on the stack, restoring it to its former size. Here, this instruction is implemented by retrieving the previously stored information on the size of the stack and popping from the stack until the target size is reached.

The instruction BREAK_LOOP works exactly the same, except that after executing it, the program jumps to the label specified during the setup. This allows the program to leave the loop from arbitrary points within it.

When encountering a for-iterator, we retrieve the counters for the current and maximum iterations from the logged iterators. If the maximum number of iterations is not yet reached, a new value depending on the iterable object is created and added to the stack. Whenever the FOR_ITER instruction is reached, the corresponding iterable is always on top of the stack and is not removed when a value is taken from it. The value's name is again constructed using the name of the iterable followed by an index — popped from the list of subscripts – in brackets.

When the loop has ended, i.e. the iteration counter has reached the recorded maximum, the iterable object is removed from the stack and the execution resumes at the label given as the argument of the FOR_ITER instruction. The iterator's current iteration counter is also reset to zero to allow the loop to be executed again.

3.4 Graph Drawing

This step converts the dependency dictionary generated by the stack machine into a graph object which is then written to a file in the Graphviz DOT language [1]. This process starts with a Value object (usually the return value of the program). Whenever a value is added to the graph, it is marked as visited. When a dependency with this marker is encountered, the algorithm does not recurse into that value's dependencies.

```
1 x = 4
2 y = 5
3
4 y = y * x
5 x = x + y
6 (x, y)
```

Listing 3.8: Example calculation

As an example, we will have a look at the code in Listing 3.8. The dependency dictionary generated during the second execution is shown in Listing 3.9.

The first node of the graph is created for the return value. Here, this is (x, y), resulting in the graph in Figure 3.2.

Then all of the value's recorded dependencies are visited. In this case, these dependencies are ('x', 1) and ('y', 1). For each dependency a node is added to the graph and connected to the value's node, this results in the intermediate graph shown in Figure 3.3.

```
1 {'_return_value': Value(name = (x, y), version = 0),
2  'dependencies': {'(x, y)': [{'named': {('y', 1), ('x', 1)}, 'unnamed': set()}],
3                   'x': [{'named': set(), 'unnamed': {Value(value = 4)}},
4                          {'named': {('y', 1), ('x', 0)}, 'unnamed': set()}],
5                   'y': [{'named': set(), 'unnamed': {Value(value = 5)}},
6                          {'named': {('x', 0), ('y', 0)}, 'unnamed': set()}]}}
```

Listing 3.9: Dependency dictionary for the calculation in Listing 3.8

Figure 3.2: Graph of return value (Step 1)

This process is then recursively used on each of the dependencies. The node x receives a connection to the already existing node for y, since it represents version 1 of y which is referenced from x. The second dependency of `('x', 1)` is version zero of the same value. Version one of y depends on version zero of both x and y. Now we have the graph in Figure 3.4.

The versions of x and y now encountered by the recursion only reference unnamed values, which are 4 and 5, respectively. The final graph is shown in Figure 3.5.

The generated data structure is now written to a file in the DOT language [1]. The resulting file is then simplified through transitive reduction using the `tred` tool included in the Graphviz package. Transitive reduction works, by removing direct connections from a graph, which are implied by indirect connections.

In our example, the connection from the return value (x, y) to y is already implied by the connections from (x, y) to x and from there to y. The same goes for the connection between the different versions of x, which is implied by the connection $x_1 \to y_1 \to x_0$. The reduced graph then looks like Figure 3.6.

It is important to note, that while this reduction, especially in larger programs, significantly simplifies the graph, it also takes away information. While the connection $(x, y) \to y$ in our example can logically be reconstructed from the reduced graph, the dependency of x_1 on x_0 cannot. If we changed line 5 of Listing 3.8 from `x = x + y` to `x = y`, the program would produce a different dependency graph, but its reduced version would look the same.

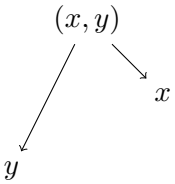


Figure 3.3: Graph of return value and its direct dependencies (Step 2)

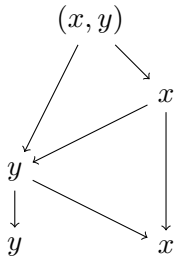


Figure 3.4: Dependency graph of return value (Step 3)

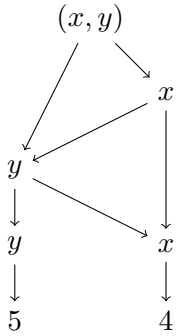


Figure 3.5: Complete dependency graph of return value (Step 4)

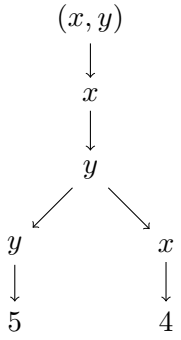


Figure 3.6: Reduced dependency graph of return value (Step 5)

4 Examples

4.1 Maximum of two values

In the following, we will have a look at the program given in Listing 4.1 and how it is analyzed throughout the steps of the implementation described previously.

```
1 def max(a, b):
2     if a >= b:
3         res = a
4     elif a < b:
5         res = b
6     return res
7
8 max(3, 4)
```

Listing 4.1: Example program taking the maximum of two values

In the instrumentation step, two conditional jumps are encountered in the bytecode. The first stems from the if condition in line 2, the second from the else-if branch in line 4. The condition in the bytecode is calculated by loading the values a and b and then applying the given comparison operation on them as can be seen in Listing 4.2. The full version of the bytecode can be found in the appendix under Listing 6.4. After this, the interpreter finds the instruction for the conditional jump POP_JUMP_IF_FALSE.

```
4     LOAD_FAST, a
5     LOAD_FAST, b
6     COMPARE_OP, >=
7     POP_JUMP_IF_FALSE, <byteplay3.Label object at 0x1011094d0>
```

Listing 4.2: If branch in original bytecode

As this instruction is reached by the instrumentation, bytecode is inserted which will later store the boolean value, which lies on top of the stack just before the jump decision is made, into the list of jump conditions. The resulting code for the first if branch is seen in Listing 4.3. The complete instrumented bytecode can be found in Listing 6.5 in the appendix. The newly inserted lines 7–13 duplicate the jump condition (7), load the log_boolean method of the logging object (9–10) and the name of the module (max, 8), put these arguments into order (ROT_THREE, 11), call the method (12) and remove its return value, which is None, from the stack, leaving it in the same state as after line

```
4    LOAD_FAST, a
5    LOAD_FAST, b
6    COMPARE_OP, >=
7    DUP_TOP
8    LOAD_CONST, max
9    LOAD_GLOBAL, logging
10   LOAD_ATTR, log_boolean
11   ROT_THREE
12   CALL_FUNCTION, 2
13   POP_TOP
14   POP_JUMP_IF_FALSE, <byteplay3.Label object at 0x1012b5510>
```

Listing 4.3: If branch in instrumented bytecode

6. For the else-if branch, the same code is inserted after the comparison operation that calculates $a < b$.

As the instrumented bytecode is executed, the program reaches both if-branches resulting in the boolean list [False, True]. The program is then compiled again to obtain the original bytecode, which is then executed using the self implemented stack machine.

Each time a conditional jump is encountered, a new value with a label beginning with COND_JUMP and ending with the jump's target label is created. This value depends on all values integral to calculating the jump condition. When a jump is *not* made, i.e. if we enter the block of the condition, this new value is added to the so called *block dependencies*. In the following block, all calculated values are now dependent on this jump condition, as their calculations would not have been executed — or at least not in this way — if the block had been skipped.

In our example, when execution reaches the POP_JUMP_IF_FALSE instructions, the first value is popped of the boolean list, to decide whether the jump is made or not. Since the first value in our list is False the first alternative is skipped. When the second jump is reached, the value True is popped of the list. Therefore the interpreter enters (or rather: does not skip) this block . Now, after the assignment res = b is made, res does not only depend on b but also on the COND_JUMP value, which in turn depends on both a and b.

Finally, the resulting dependency dictionary is converted into the graph seen in Figure 4.1. Through the transitive reduction applied to the graph, the edge connecting res

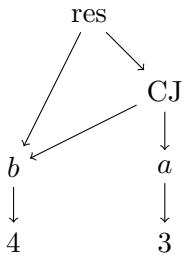


Figure 4.1: Dependency graph for function call max(3, 4), the conditional jump is abbreviated as CJ

directly to b is removed, since it is implied through the connection res $\rightarrow$ CJ $\rightarrow b$. This results in the reduced graph shown in Figure 4.2.

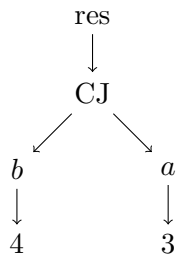


Figure 4.2: Reduced dependency graph for function call max(3, 4)

4.2 Query

This example implements an SQL join-query in Python using nested while loops. The query, shown in Listing 4.4, and the input data, are taken from Cheney et al. [2, p. 383]. The Python source code is displayed in Listing 4.5. The full source, which includes the

```
1 SELECT e.name, a.phone
2   FROM Agencies a, ExternalTours e
3  WHERE a.name = e.name
4    AND e.type = 'boat'
```

Listing 4.4: SQL query adapted from Cheney et al. [2]

input data, can be found in Listing 6.3 in the appendix.

In this instance, the graph drawing Python script was passed a flag to skip conditional jumps which simplifies the resulting graph. Skipping the conditional jumps works by forwarding incoming connections to the conditional jump's node to all nodes the jump would have connections to.

The graph shown in Figure 4.3 was also simplified by manually editing the dependency dictionary to remove the building of the input data. An unedited, non-reduced version can be found in the appendix under Figure 6.1.

At the top of the graph, one can see that after being initialized as the empty list `[]`, the output variable res is updated two times. These updates happen each time a tuple is added to the output in line 46. The first output tuple depends on the first element in the input relation a (`a[0]`) and the third element of the input relation e (`e[2]`). The second output tuple stems from the input tuples `a[1]` and `e[3]`.

```
def myquery(a, e):
    res = []
    i = 0
    lengtha = len(a)
    lengthe = len(e)
    while i<lengtha:
        k = 0
        while k<lengthe:
            ta = a[i]
            te = e[k]
            name_a = ta["name"]
            name_e = te["name"]
            if name_a == name_e:
                type_e = te["type"]
                if type_e == "boat":
                    t = {}
                    t["name"] = name_e
                    t_p = ta["phone"]
                    t["phone"] = t_p
                    res.append(t)
            k =k+1
        i = i+1
    return res
```

Listing 4.5: Python implementation of the SQL query

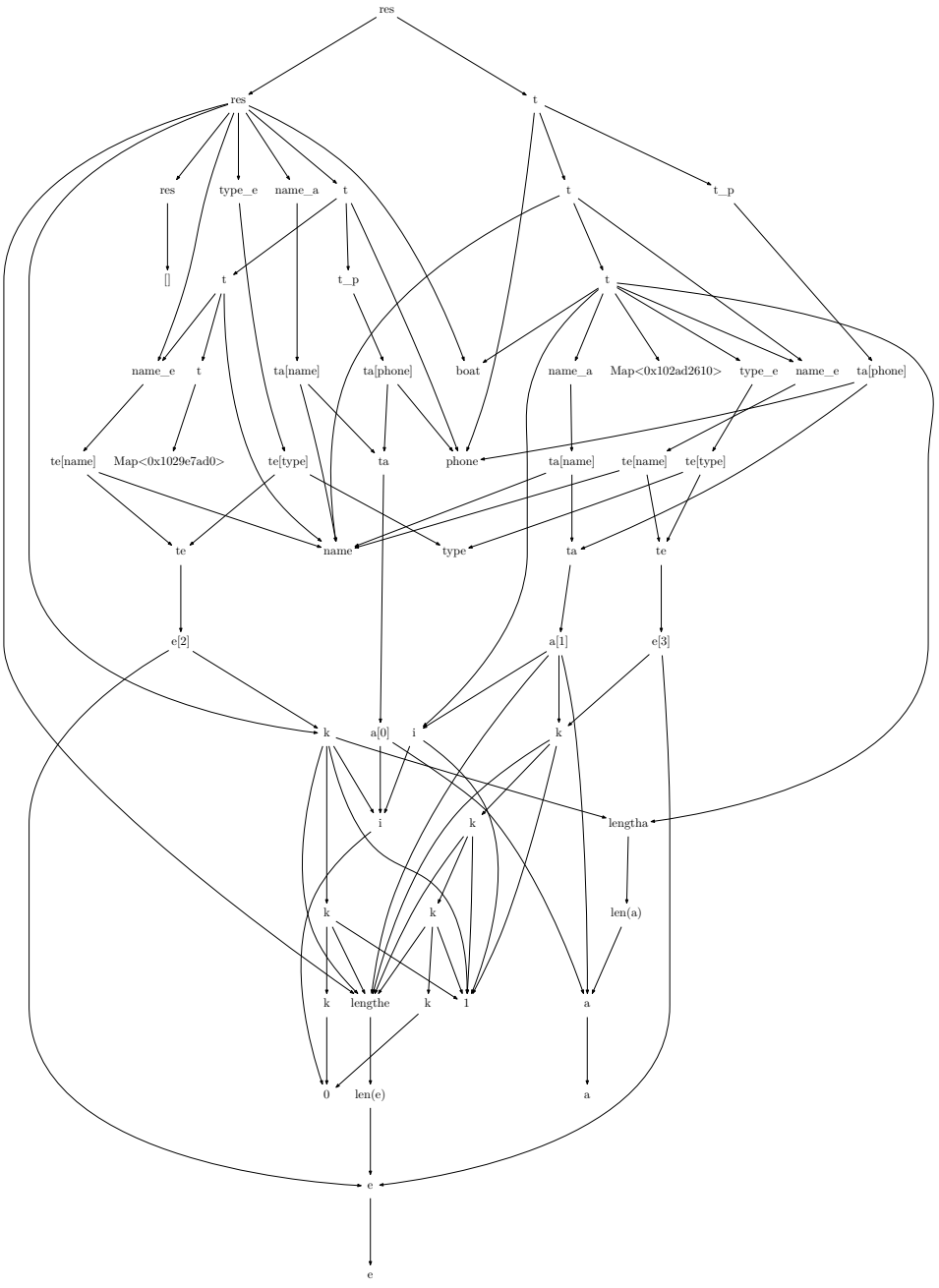


Figure 4.3: Reduced and manually simplified dependency graph for SQL query

5 Conclusion and Future Work

5.1 Points for Improvement

The implementation poses certain limitations which could possibly be resolved through further development.

5.1.1 Built-in Functions

Since the concept is based on bytecode operations, built-in functions of the Python interpreter[1] cannot be properly tracked, as they are implemented in C. This implementation therefore assumes that the result of a call to a built-in function depends on all of its input-variables. The same applies to methods of built-in data types — such as lists, dictionaries and tuples — which are also written in C. Resolving this problem would probably require extending the implementation of the CPython interpreter, making it a task that could not be covered in this work due to time limitations.

5.1.2 Support for Python's Functionality

Also due to time constraints, several aspects of the CPython interpreter were not implemented in the stack machine. This includes importing of modules using variations of the **import** statement, definition of classes and instantiation of objects. The implementation of these features would however certainly be possible with this approach.

5.1.3 Nested For-iterators

To support nested iterations, a for-iterator is (re-)set to zero, before the loop is entered. This happens during the execution of the instrumented bytecode. Because of this, only the last number of iterations is recorded. This might produce incorrect results, if an inner loop is executed a different number of times in any iteration of the outer loop. For example, if the iterable object for the inner loop changes in length.

It is notable that this makes for-iterators unsafe. If, for example, the inner loop contains an if-statement, not enough or too many values could be removed from the list of booleans. The results could be unpredicted control flow or even errors in the interpreter, when trying to remove a value from an empty list.

This problem could be solved by keeping a separate maximum-iteration counter for each iteration of the outer loop.

[1] For a list, see: `http://docs.python.org/3/library/functions.html#built-in-functions`

5.1.4 If Else Branches

If branches that end in an else statement proved as a further problem. Since there is no conditional jump when an else statement is encountered, there is no condition that can be related to the execution of the else branch, even though the execution of the branch actually depends on the failure of all previous if and else-if conditions.

5.1.5 Segmentation Faults

For unknown reasons, the implementation produces segmentation faults on certain input programs. Also, the modules `instrument.py` and `execute.py` produce segmentation faults and/or abort traps, when a certain, unnecessary input is removed. A reason or solution for this behaviour has not yet been found.

5.2 Conclusion

Despite these limitations, the presented implementation provides an interesting approach to program analysis. Provided the input program does not exceed a certain level of complexity, even an inexperienced user could reason about the data flow of the program. It is also important to note, that this work and the graphs generated as its output are merely intermediate steps in a project of a larger scale.

The dependency dictionary created during the execution of the program in the self-implemented stack machine, could, for example, be used to visualize the resulting information in a different way. It would also be possible to extend the stack machine to differentiate dependency relations of values more precisely. An example for this are subscript operations. By recording the collection and its accessed index separately from the rest of the dependencies, a more fine-grained analysis could be performed. This additional information could then be used to visualize the subscript operation differently from other dependency relations.

6 Appendix

Note on Byteplay

The Byteplay module is used in the version by Rumed [5] which provides Python 3 compatibility for the original library written by Yorav-Raphael [6]. An adjustment to work with function definitions in Python version 3.3 was made in lines 140–155. Up until version 3.2 function definitions worked as described in Listing 6.1. In Python 3.3,

```
1   LOAD_CONST <code>
2   MAKE_FUNCTION None
```

Listing 6.1: Function definitions in Python bytecode up until version 3.2

however the attribute __qualname__ was added to functions and classes. This qualified name is now loaded before the call to MAKE_FUNCTION as can be seen in Listing 6.2.

```
1   LOAD_CONST <code>
2   LOAD_CONST <qualified_name>
3   MAKE_FUNCTION None
```

Listing 6.2: Function definitions in Python bytecode since version 3.3

Full Code and Graphs

```
1  # example based on:
2  # Cheney et al.
3  # Provenance in Databases
4
5  # written by Tobias Müller
6  # relational data
7  a = [
8          {"name": "BayTours",    "based_in": "San Francisco",    "phone": "415-1200"},
9          {"name": "HarborCruz",  "based_in": "Santa Cruz",       "phone": "831-3000"}
10     ]
11
12 e = [
13         {"name": "BayTours",    "destination": "San Francisco", "type": "cable car",    "price": 50},
14         {"name": "BayTours",    "destination": "Santa Cruz",    "type": "bus",          "price": 100},
15         {"name": "BayTours",    "destination": "Santa Cruz",    "type": "boat",         "price": 250},
16         {"name": "HarborCruz",  "destination": "Monterey",      "type": "boat",         "price": 200},
```

```
17          {"name": "HarborCruz",  "destination": "Carmel",        "type": "train",          "price": 90}
18      ]
19
20  # SQL equivalent of python code below
21
22  # SELECT e.name, a.phone
23  #   FROM Agencies a, ExternalTours e
24  #  WHERE a.name = e.name
25  #    AND e.type = 'boat'
26
27  def myquery(a, e):
28      res = []
29      i = 0
30      lengtha = len(a)
31      lengthe = len(e)
32      while i<lengtha:
33          k = 0
34          while k<lengthe:
35              ta = a[i]
36              te = e[k]
37              name_a = ta["name"]
38              name_e = te["name"]
39              if name_a == name_e:
40                  type_e = te["type"]
41                  if type_e == "boat":
42                      t = {}
43                      t["name"] = name_e
44                      t_p = ta["phone"]
45                      t["phone"] = t_p
46                      res.append(t)
47              k =k+1
48          i = i+1
49      return res
50
51  myquery(a, e)
52
53  # python result:
54  # [
55  #  {'phone': '415-1200', 'name': 'BayTours'},
56  #  {'phone': '415-1200', 'name': 'BayTours'},
57  #  {'phone': '831-3000', 'name': 'HarborCruz'}
58  # ]
```

Listing 6.3: Python implementation of the SQL query (full source)

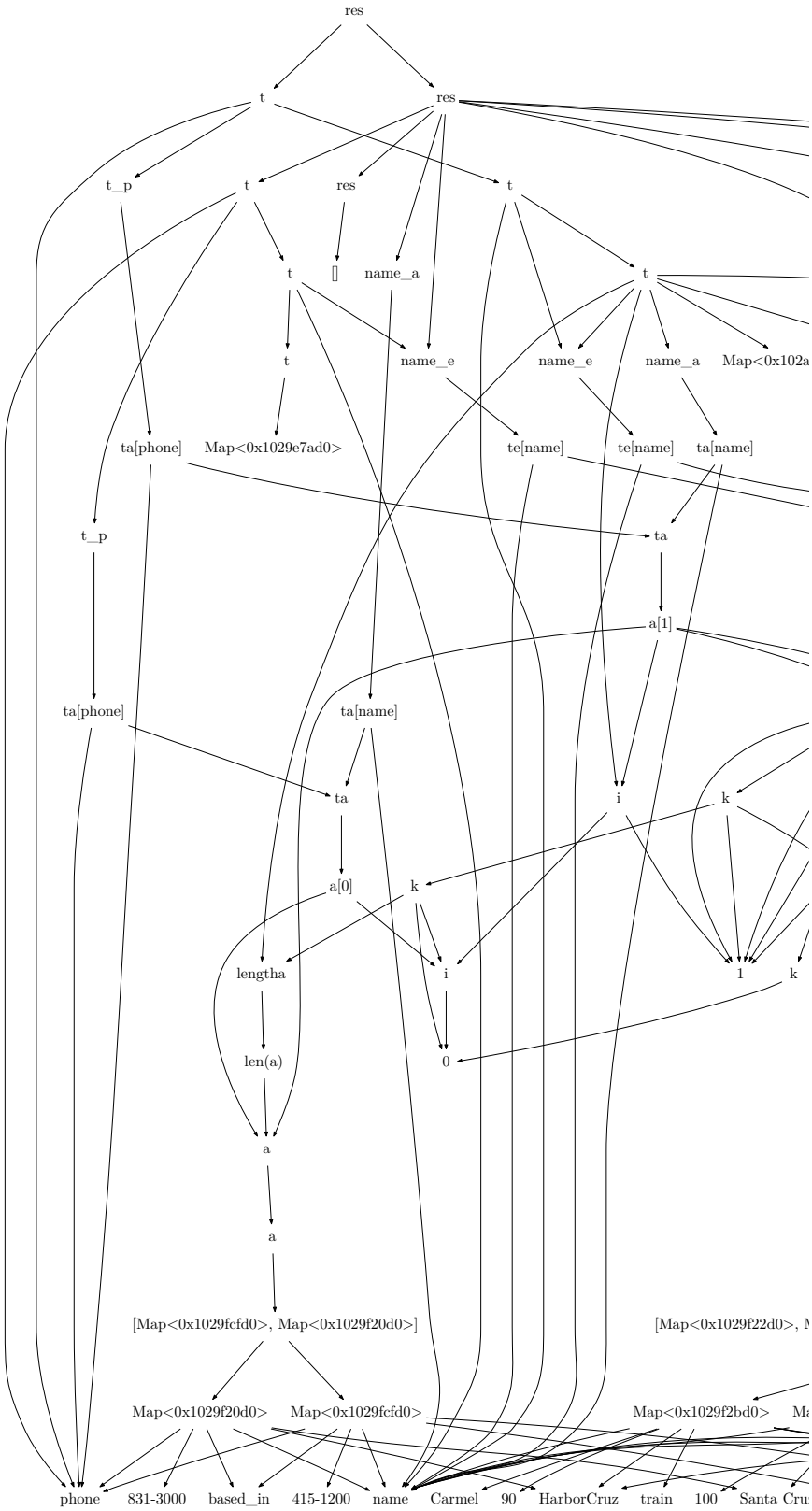
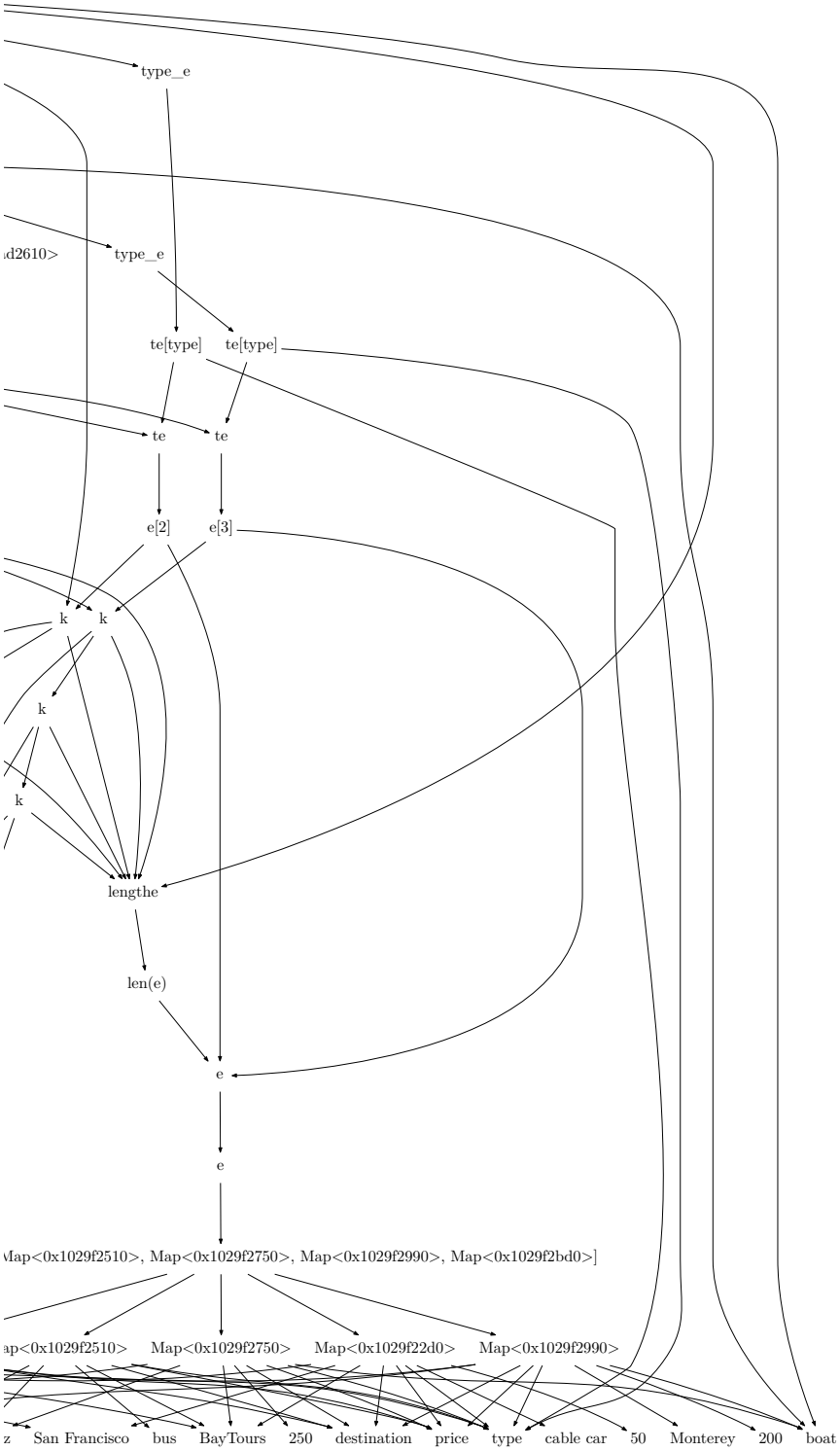


Figure 6.1: Full dependency graph for SQL query


type_e
d2610>
type_e
te[type]
te[type]
te
te
e[2]
e[3]
k
k
k
k
lengthe
len(e)
e
e
Map<0x1029f2510>, Map<0x1029f2750>, Map<0x1029f2990>, Map<0x1029f2bd0>]
ap<0x1029f2510>
Map<0x1029f2750>
Map<0x1029f22d0>
Map<0x1029f2990>
z
San Francisco
bus
BayTours
250
destination
price
type
cable car
50
Monterey
200
boat

```
1  <class 'byteplay3.SetLinenoType'>, 1
2  LOAD_CONST, <byteplay3.Code object at 0x101109490>
3      <class 'byteplay3.SetLinenoType'>, 2
4      LOAD_FAST, a
5      LOAD_FAST, b
6      COMPARE_OP, >=
7      POP_JUMP_IF_FALSE, <byteplay3.Label object at 0x1011094d0>
8      <class 'byteplay3.SetLinenoType'>, 3
9      LOAD_FAST, a
10     STORE_FAST, res
11     JUMP_FORWARD, <byteplay3.Label object at 0x101109510>
12     <byteplay3.Label object at 0x1011094d0>
13     <class 'byteplay3.SetLinenoType'>, 4
14     LOAD_FAST, a
15     LOAD_FAST, b
16     COMPARE_OP, <
17     POP_JUMP_IF_FALSE, <byteplay3.Label object at 0x101109510>
18     <class 'byteplay3.SetLinenoType'>, 5
19     LOAD_FAST, b
20     STORE_FAST, res
21     JUMP_FORWARD, <byteplay3.Label object at 0x101109510>
22     <byteplay3.Label object at 0x101109510>
23     <class 'byteplay3.SetLinenoType'>, 6
24     LOAD_FAST, res
25     RETURN_VALUE
26 LOAD_CONST, max
27 MAKE_FUNCTION, 0
28 STORE_NAME, max
29 <class 'byteplay3.SetLinenoType'>, 8
30 LOAD_NAME, max
31 LOAD_CONST, 3
32 LOAD_CONST, 4
33 CALL_FUNCTION, 2
34 POP_TOP
35 LOAD_CONST
36 RETURN_VALUE
```

Listing 6.4: Bytecode for the program in Listing 4.1

```
1  <class 'byteplay3.SetLinenoType'>, 1
2  LOAD_CONST, <byteplay3.Code object at 0x1012b54d0>
3      <class 'byteplay3.SetLinenoType'>, 2
4      LOAD_FAST, a
5      LOAD_FAST, b
6      COMPARE_OP, >=
7      DUP_TOP
8      LOAD_CONST, max
9      LOAD_GLOBAL, logging
10     LOAD_ATTR, log_boolean
11     ROT_THREE
12     CALL_FUNCTION, 2
13     POP_TOP
14     POP_JUMP_IF_FALSE, <byteplay3.Label object at 0x1012b5510>
```

```
15      <class 'byteplay3.SetLinenoType'>, 3
16      LOAD_FAST, a
17      STORE_FAST, res
18      JUMP_FORWARD, <byteplay3.Label object at 0x1012b5550>
19      <byteplay3.Label object at 0x1012b5510>
20      <class 'byteplay3.SetLinenoType'>, 4
21      LOAD_FAST, a
22      LOAD_FAST, b
23      COMPARE_OP, <
24      DUP_TOP
25      LOAD_CONST, max
26      LOAD_GLOBAL, logging
27      LOAD_ATTR, log_boolean
28      ROT_THREE
29      CALL_FUNCTION, 2
30      POP_TOP
31      POP_JUMP_IF_FALSE, <byteplay3.Label object at 0x1012b5550>
32      <class 'byteplay3.SetLinenoType'>, 5
33      LOAD_FAST, b
34      STORE_FAST, res
35      JUMP_FORWARD, <byteplay3.Label object at 0x1012b5550>
36      <byteplay3.Label object at 0x1012b5550>
37      <class 'byteplay3.SetLinenoType'>, 6
38      LOAD_FAST, res
39      RETURN_VALUE
40  LOAD_CONST, max
41  MAKE_FUNCTION, 0
42  STORE_NAME, max
43  <class 'byteplay3.SetLinenoType'>, 8
44  LOAD_NAME, max
45  LOAD_CONST, 3
46  LOAD_CONST, 4
47  CALL_FUNCTION, 2
48  POP_TOP
49  LOAD_CONST
50  RETURN_VALUE
```

Listing 6.5: Instrumented bytecode for the program in Listing 4.1

List of Figures

Listings

Bibliography

[1] AT&T Labs Research. The DOT Language, March 2014.
http://www.graphviz.org/content/dot-language.

[2] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Found. Trends databases*, 1(4):379–474, Apr. 2009. ISSN 1931-7883. doi: 10.1561/1900000006.
http://dx.doi.org/10.1561/1900000006.

[3] Python Software Foundation, March 2014.
http://www.python.org.

[4] Python Software Foundation. Python bytecode instructions, February 2014.
http://docs.python.org/3/library/dis.html#python-bytecode-instructions.

[5] D. Rumed. byteplay for Python 3, June 2013.
https://github.com/serprex/byteplay.

[6] N. Yorav-Raphael. byteplay - Python bytecode assembler-disassembler, September 2010.
https://code.google.com/p/byteplay.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Ort, Datum, Unterschrift