# TAPY

**Type Analysis for Python**

**Jesper Lindstrøm Nielsen** - *jesper.jln@gmail.com*
**Christoffer Quist Adamsen** - *christofferqa@gmail.com*
**Troels Leth Jensen** - *troelslethjensen@gmail.com*

**Aarhus University**
**May 3, 2013**

# Abstract

TODO

# 1 Introduction

Python is a dynamically typed, general purpose programming language that supports both object-oriented, imperative and functional programming styles. As opposed to most other programming languages Python is an indented language, with the intention to allow programmers to write more concise code.
Because of its dynamic nature and little tool support it can be difficult to develop and maintain larger programs. In this report we present our work towards developing a conservative type analysis for Python in Scala.

# 2 Limitations

### 2.0.1 Numbers

The size of an integer in Python depends on the system your on. If your running python on a x64 machine you will be able to create integers in the interval $[-2^{63} - 2; 2^{63} - 1]$, where on a x86 machine your integers would be limited to the interval $[-2^{31} - 2; 2^{31} - 1]$. In our analysis we assume that our target machine is a x86 and the integers is then limited to 32-bits. This is also described in The Python Language Reference data model[1] in section 3.2.

### 2.0.2 Language functionality

Python is a general purpose programming language and quite popular, it has been developed and extended and holds a lot of functionality, some of these functionalities is not used that often. In our analysis we have decided to exclude some of these language functionalities;

- With statements
- Lambda expressions
- Generator expressions
- Yield expressions

### 2.0.3 Magic methods

Python uses magic methods to give the developer flexibility to do operation overloading on custom classes. These methods makes simple operations such as binary operations and unary operations hard to emulate in the control flow graph since its not just an operation but could actually be a method-call with potentially side-effects.
We have choosen to focus on the magic methods that is used when properties on objects is read. These methods is called __getattribute__ and __getattr__. A complete list and description on the magic methods can be found in The Python Language Reference data model[1] in section 3.4.

# 3 Dynamic features in Python

As mentioned Python is a dynamically typed language, and therefore has a lot in common with e.g. JavaScript. In this section we present some of its interesting dynamic features, together with a bunch of common runtime errors.

Classes are declared using the `class` keyword, supports multiple inheritance and can be modified further after creation. The code in 1 declares an empty `Student` class that inherits from `Person`, which in turn inherits from `object`. In line 14 a function `addGrade` is added to the `Student` class and in line 16 the attribute grades is set to an empty dictionary on the $s1$ object. Therefore we can call the `addGrade` function on the `s1` object without getting a runtime error. However, since we forgot to set the grades attribute on the $s2$ object, we get the following runtime error from line 19: `AttributeError: 'Student' object has no attribute 'grades'`.

Notice that the receiver object is given implicitly as a first argument to the addGrade function. In case we had forgot to supply the extra formal parameter `self`, the following runtime error would result from line 18: `TypeError: addGrade() takes exactly 2 arguments (3 given)`.

Another interesting aspect with regards to parameter passing is that Python supports unpacking of argument lists. For instance we could have provided the arguments to the addGrade function in line 18 by means of a dictionary instead: `s1.addGrade(**{ 'course': 'math', grade: 10})`.

```
1  class Person(object):
2    def __init__(self, name):
3      self.name = name
4  class Student(Person):
5    pass
6  s1 = Student('Foo')
7  s2 = Student('Bar')
8  def addGrade(self, course, grade):
9    self.grades[course] = grade
10 Student.addGrade = addGrade
11 s1.grades = {}
12 s1.addGrade('math', 10)
13 s2.addGrade('math', 7)
```

Listing 1: Magic method example in Python

Unlike as in JavaScript, we wouldn't be able to change line 16 into `s1['grades'] = {}`. This would result in the following error: `TypeError: 'Student' object does not support item assignment`, while trying to access `s1['grades'] would result in the following error: TypeError: 'Student' object has no attribute '__getitem__'`. Instead it is possible to call the built-in functions `getattr(obj,attr)` and `setattr(obj,attr,val)`.

But Python also allows the programmer to customize the behavior when indexing into an object by supplying special functions `__setitem__` and `__getitem__`, giving the programmer much more control. As an example consider the new Student class in 2. With this implementation we could set the grades attribute as in JavaScript: `s1['grades'] = {}`, and e.g. get the grade of `s1` in the math course by calling `s1.math` or `s1['math']`.

```
1  class Student(Person):
2    def __getitem__(self, name):
3      return self.grades[name]
4    def __setitem__(self, name, val):
5      setattr(self, name, val)
6    def __getattr__(self, name):
7      if name in self.grades:
8        return self.grades[name]
9      else:
10       return "<no such grade>"
```

Listing 2: Magic method example in python

# 4 Monotoneframework implementation

## 4.1 Lattices

In order to make our lives easier when constructing the final analysis lattice, we have implemented several common lattice structures as type generic classes. These common lattice structures include, among others, a map- and product lattice. In this section we will briefly go over the implementation decisions made for these structures.

Using classic OOP principles each of these compound structures decide the ordering of their elements by delegating to the underlying lattices in a point wise fashion, e.g. the product lattice has two underlying lattices, one for each element and thus the ordering is decided by comparing the first element in the pair in the context of the first lattice and similarly for the second element.

The map lattice has received a couple changes from the native implementation to make it usable in more cases. The first change was to interpret an unbound key value, k, to be a mapping from k to the bottom element of the underlying lattice. In some use cases, such as a functional approach to intra procedural static analysis, the map lattice will have a huge amount of keys. Requiring all of these to be bound to some value in the map is superfluous. This invariant is hidden completely in the lattice class because you can't manipulate the lattice element directly, so when trying to lookup an unbound key, the lattice simply constructs a fresh bottom value.

Since we now have a way to not bind every value from the key set, we are also able to change the constructor from the straightforward approach s: Set[T], l: Lattice[S] into a more general s: T, l: Lattice[S] (where S and T are type arguments). The straightforward approach has to computing the entire key set before you are able to instantiate the map lattice, but since the key set in itself might be exponentially large that wouldn't be practical. The downside to this change is, that since map lattice has no way to know the intended key set, there is no way to construct the top element of the lattice.

The top element of the lattice is useful for when you want to give up in the analysis, so to fix we instrumented the map lattice with a new top element, in a similar fashion to how the sink lattice instruments the underlying lattice with a new bottom element.

<powersub/super>

## 4.2 Constraints

Being in a functional programming language an easy to work with representation of the constraints is anonymous lambdas with the type E -> E, where E is the type of the elements in the analysis lattice. Each constraint captures the node it was made from in its closure, so it is able to lookup the needed information in the lattice element.

This approach follows the notation very nicely, and as such makes the implementation a simple task when the constraints have been formulated formally.

# 5 Control Flow Graph construction

In this section we give examples of how we inductively generate the control flow graph of some essential language features. During this we will present the control flow graph nodes that we use, together with their semantics.
In Python, both for and while loops has an else branch, which is executed if the loop terminates normally (i.e. not using `break`).

```
"for" target_list "in" expression_list ":" suite
["else" ":" suite]
```

Listing 3: For-loop syntax according to the Python Language Reference

```
for x in [1,2,3]:
  print x
else:
  print "not a break'ed exit"
```

Listing 4: For-loop example

First, `expression_list` is evaluated. This should yield an iterable object, such that an iterator object can be created. Now for each element of the iterator the body of the for loop is evaluated once. As mentioned, if an else block is provided to the loop, it is evaluated when all iterations are done, and the iteration did not stop because of a break statement.
To simulate the evaluation sequence of such a for loop in our control flow graph, we need to take a look at the iterator object: to get the next element from a Python iterator object the next method is used. This method returns the next

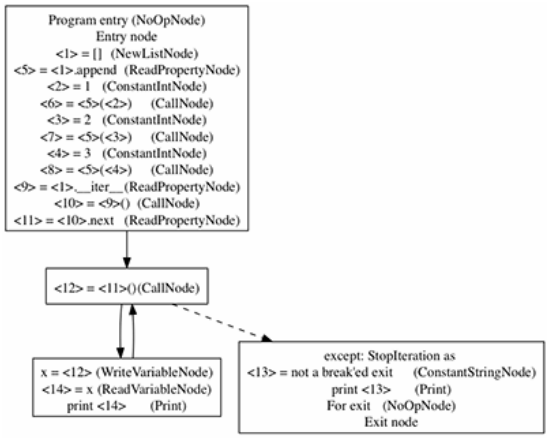


**Figure 5.1:** Control Flow Graph for the for-loop example 4

element until the iteration is done and finally raises a `StopIteration` exception. The CFG for the for-loop example 4 can been seen in figure 5.1.

For a while loop we generate the control flow graph in figure 5.2.
$CFG_{cond}$ is the control flow graph that results from the condition. If the condition is the boolean `True`, $CFG_{cond}$ will be the control flow graph consisting of a single node, namely *ConstantBooleanNode*. Inspired from TAJS, Type Analyzer for JavaScript, our *ConstantBooleanNode* holds a result register ($reg_{cond}$ in figure 5.3) together with the actual constant value, `True` in this case.
The other nodes *ConstantIntNode*, *ConstantFloatNode*, *ConstantLongNode*, *ConstantComplexNode*, *ConstantStringNode*, *ConstantNoneNode*, *NewListNode*, *NewDictionaryNode* and *NewTupleNode* work in a similar way to *ConstantBooleanNode*.
The motivation for introducing registers is that a single expression as e.g. `s1.addGrade('math', 10)` is evaluated in several steps.

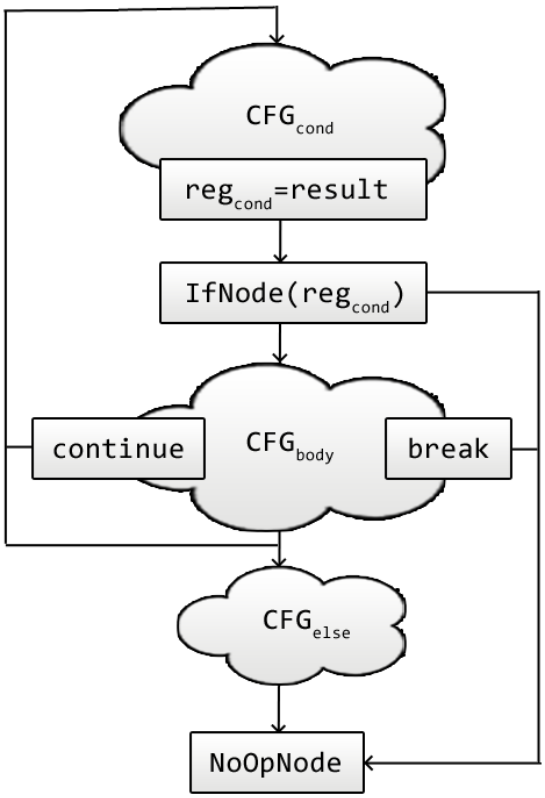


**Figure 5.2:** Control Flow Graph for a while-loop

First the function `addGrade` is looked up in the class of `s1`. This is done in the control flow graph using *ReadPropertyNode*, which holds a result register, a base register, i.e. the register where to find the object, and the name of the property to look up. Similar nodes include *ReadVariableNode* and *ReadIndexableNode*.

Secondly, each argument given to the function is evaluated, and finally the actual call is done. Calls in the control flow graph is modeled using the *CallNode*, which holds a result register, a function register and a list of arguments registers.
Thus the expression `s1.addGrade('math', 10)` will result in the control flow graph found in figure 5.3 (where the numbers to the left represent the result registers of the nodes).

So far we have primarily been concerned with putting constants into registers and reading e.g. variables. In order to support writing we have three different nodes: *WriteVariableNode*, *WritePropertyNode*, and *WriteIndexableNode*. Besides holding a value register, i.e. the register where to find the value being written, *WriteVariableNode* contains the name of the variable being written to, *WritePropertyNode* contains a base register and the property being written to, and *WriteIndexableNode* contains a base and property register (the latter has a register for the property be-

Tilfoeje afsnit om __getattribute__ og __getattr__, samt hvordan disse haandteres.

cause it is not constant, for instance we could write something like the following: `dict[getKey()] = aValue`, whereas property in `obj.property = aValue` must be a string).

## 5.1 Handling exceptions

In order to handle the flow caused by exceptions we use two different kinds of edges in the control flow graph. A solid edge indicates normal flow, and a dashed edge indicates exception flow.
In Python exceptions can be caught using a try-except-else-finally block. An except block can be annotated with a number of types, and each try-except-else-finally block may contain an arbitrary number of except blocks. As usual, the else block is entered in case of a normal exit, i.e. when no exceptions were raised inside the try block.
The AST provided by the Jython parser has been normalized from a try-except-else-finally block into a try-finally block, which contains a try-except-else block in its try block.

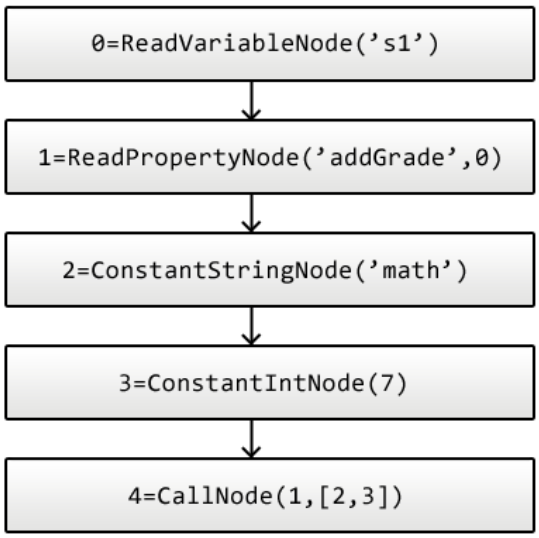


**Figure 5.3:** Control Flow Graph for a call example

For instance the code in the first example 5 below is transformed into the code in the next example 6:

```
1  try:
2    <try-stms>
3  except Foo:
4    <except-foo-stms>
5  except:
6    <except-stms>
7  else:
8    <else>
9  finally:
10   <finally>
```

Listing 5: A try-except-else-finally example before convertion

```
try:
  try:
    <try-stms>
  except Foo:
    <except-foo-stms>
  except:
    <except-stms>
  else:
    <else-stms>
finally:
  <finally-stms>
```

Listing 6: A try-except-else-finally example after convertion

Inductively, CFG's for the statement lists <`try-stms`> ($CFG_{try}$), <`except-foo-stms`> ($CFG_{except-foo}$), <`except-stms`> ($CFG_{except}$), <`else-stms`> ($CFG_{else}$), and <`finally-stms`> are created.
The CFG for the finally block is then cloned into three duplicates ($CFG_{finally-normal}$, $CFG_{finally-handled-exc}$, $CFG_{finally-unhandled-exc}$). The purpose is to have one finally block for each of the following cases:

1. when no exceptions occur during the try block,
2. when an exception is raised and caught by one of the surrounding except blocks, and no exception is raised from inside that except block, and
3. when an exception is raised but not caught, which is the case when a) an exception is raised from the try block and no except blocks handles this particular exception, b) an except block catches an exception raised by the try block, but then raises a new exception on its own.

In particular, it is important that the finally block for handling case (3), i.e. $CFG_{finally-unhandled-exc}$, is not connected to the exit node of the try-except-else-finally block. Instead, it should be connected to its nearest surrounding except block (if any), or no except block at all (indicating that the program crashes with a runtime error because of an unhandled exception).
In the following sections we present the way we generate the CFG of a try-except-else-finally block.

### 5.1.1 The try block

Each node in $CFG_{try}$ (that does not already have an outgoing exception edge) is connected using an exception edge to the entry node of the first except block (*), in this case the entry node of $CFG_{except-foo}$. We do not add exception edges to nodes that already have an exception edges, because this would be a loss of information:

the control flow always goes to the nearest enclosing except block in case of exceptions.
The above models that if an exception occurs during evaluation of one of the statements in a try block, then the control flow will proceed from the first except block. If there is no except blocks, each node should instead be connected using an exception edge to the entry node of its nearest surrounding except or finally block (specifically $CFG_{finally-unhandled-exc}$). However, we don't add any exception edges here; these will be added inductively because of (*) in case there are any surrounding except or finally blocks.

### 5.1.2 The except block

The entry node of each except block is connected using an exception edge to the entry node of the next except block (except for the last block, of course). Thus we make an exception edge from the entry node of $CFG_{except-foo}$ to the entry node of $CFG_{except}$.
We do this because the first except block might not catch the exception (because of the type restrictions), in which the control flow proceeds at the next except block.
Furthermore, each node inside the except block should be connected using an exception edge to the entry node of its nearest surrounding except or finally block (specifically $CFG_{finally-unhandled-exc}$). As above, this is handled inductively.
Finally, if an except block actually catches the exception, and no exceptions occur inside that except block, the control flow proceeds to the surrounding finally block (in this case, $CFG_{finally-handled-exc}$).

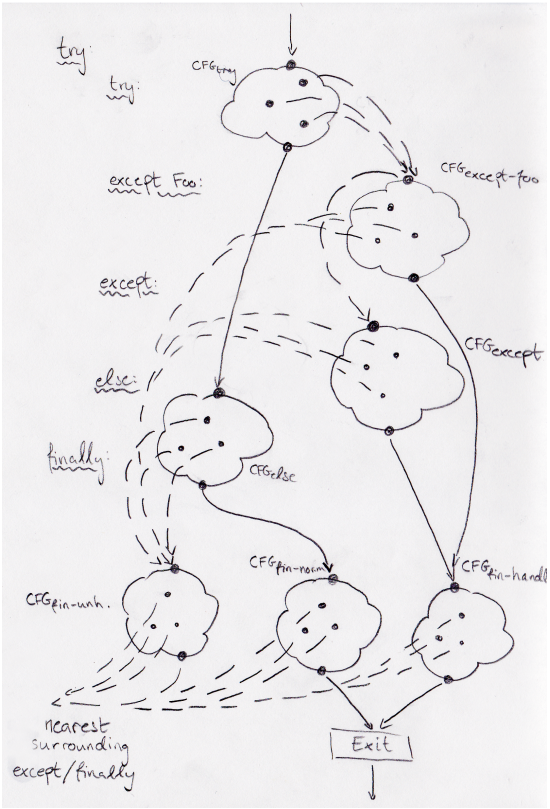


**Figure 5.4:** Control Flow Graph for try try-except example 6

### 5.1.3 The else block

If no exceptions occur, the else block should be evaluated. Thus we add a normal flow edge from the exit node of $CFG_{try}$ to the entry node of $CFG_{else}$.
Since exceptions may result from evaluating the statements in the else block, each node in $CFG_{else}$ should also be connected to the entry node of the nearest surrounding except or finally block (again, $CFG_{finally-unhandled-exc}$).
In case the evaluation of the statements in the else block does not raise any ex-

ceptions, the control flow proceeds either to the exit node of the whole try-except-else block, or in case there is a surrounding finally block, to the entry node of $CFG_{finally-normal}$.

# 6 The Analysis Lattice

Inspired by TAJS we have a lattice for abstract values, $Value$, from which we build a lattice for abstract objects, $Object$. These two lattices are the main building blocks for the lattice for abstract states, $State$. Our analysis lattice is the lattice which for each program point (i.e. for each CFG node) tells the abstract state of that program point. Furthermore the analysis lattice tells the call graph of the CFG.

### 6.0.4 Abstract Values

The concrete lattices follows below.

$$\begin{aligned} Value = \quad & Undefined \times None \times Boolean \times Integer \\ & \times Float \times Long \times Complex \times String \times P(ObjectLabel) \end{aligned}$$

The value lattice is used to tell the value of a temporary variable (see the lattice Stack), and a property on an object (see the lattice Object). The Undefined and None lattices both contains two nodes, Top and bottom. In Python when a function does not return a value, it is the constant None[2] which is returned, e.g.:

```
1 def a():
2         pass
3 a() is None // true
```

Listing 7: Constant None

Contrary to JavaScript, Python supports integers, floats, longs and complex numbers, so we have separate lattices for those. Note that $Complex = Float \times Float$, since a complex number in Python is represented using a float for the real and imaginary part, respectively[3]. The Integer lattice is defined here6.1 The Float, Long and String lattices are defined in similar ways. Finally, a value can of course also be a pointer to an object in the

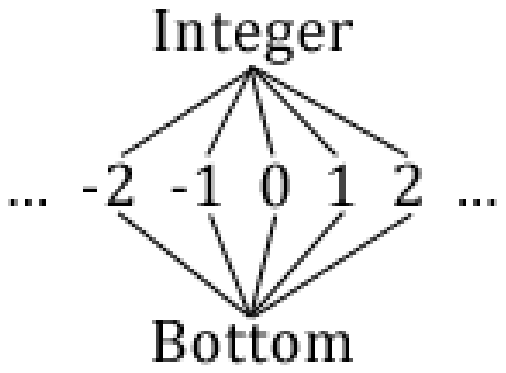


**Figure 6.1:** The integer lattice

heap, which we model in the Value lattice by having a power set[1] of object labels, $P(ObjectLabel)$.

### 6.0.5 Abstract State

We use the following lattice to model abstract state:

$$State = Heap \times Stack$$

In the following sections the $Heap$ and $Stack$ lattice will be described, but first it is necessary to look at the $Object$ lattice:

$$Object = (PropertyName \rightarrow Value) \times P(ObjectLabel^*)$$

Having made special lattices for the 'primitive' objects there is still a need to handle the more complex objects such as function objects. As with JavaScript you can augment objects with properties at runtime, so the lattice needs to accommodate this dynamic behavior, so we use a map from property names to values. For some objects it is required to track the scope in which they were defined, to model the closure they are evaluated in. Thus the object lattice is the product between ObjectValues and a scope chain modelled as a list of object labels.

### 6.0.6 The Heap

$$Heap = (ObjectLabel \rightarrow Object)$$

The heap is modelled by a map from object labels to an object value. We found it benefitial to distinquise between different types of objects but still handle them in the same way in the heap. To achieve this we made several different subclasses to the object label object, e.g. function object, which besides its name and id also holds a reference to the cfg node that is the entry node for that function.

### 6.0.7 The Stack

The $Stack$ lattice is defined as:

$$Stack = (TempVar \rightarrow Value) \times P(ObjectLabel^* \times ObjectLabel)$$

For each temporary variable, i.e. register, we specify the value of that particular temporary variable. The power set $P(ObjectLabel^* \times ObjectLabel)$ specifies the scope chain ($ObjectLabel^*$) and variable object ($ObjectLabel$). The variable object determines which object on the heap, local variable writes should we written to. For instance we will for each program have an object on the heap, that models the module/top-level script environment `__main__`[4] (this is what corresponds to the global object in JavaScript). Whenever an assignment to a variable occurs in the top-level scripting environment, e.g. `x=10`, the variable `x` is set as a property

[1] All our power sets are ordered by subset inclusion.

mapping to the integer 10 on the `__main__` object in the heap. The scope chain specifies where to look in case of e.g. reading a variable that is not present on the variable object. The following simple example can be used to illustrate this:

```
1 x = 10
2 def a():
3         print x
```

Listing 8: Scope example

For this particular Scope example**??** we will have the following objects on the heap:

1. The `__main__` object,
2. The object of the function `a`,
3. The function object of `a`, and
4. The scope object of `a` (which is an object similar to the `__main__` object, i.e. an object where local variables are written onto).

In line 3 the variable object will be set to the scope object of the function a (or more precise, its label), and the scope chain will be the list containing the `__main__` object. When the variable x is read, x is not found on the variable object, so it is looked up in the scope chain, where it is found on the `__main__` object. When there are more than one object on the scope chain, those objects are of course just traversed starting from the head of the list. In this section we try to motivate why we have three objects on the heap for each function. The object of the function a in 9 is necessary as functions are themselves objects, as in JavaScript. For instance we can set a property on a function object as line 3 in the following example illustrates:

```
1 def a():
2         pass
3 a.prop = 42
```

Listing 9: Property on function object

Thus we map the property prop to the integer 42 on the object of the function. Furthermore we map the magic property `__call__` to the function object of a. Whenever a function is declared in Python, the object of the function will have this property as this example illustrates:

```
def a():
        print "a"
a() // "a"
a.__call__ # <method-wrapper '__call__' of function object at ...>
a.__call__() # "a"
```

Listing 10: Function object and __call__ example

Thus the function object in 10 can be considered as the object of the method-wrapper. It is important to distinguish between the object of the function, and the function object, since `__call__` is not just a reference to the object of the function, as illustrated below:

```
def a():
        pass
a.__call__.prop = 10 # TypeError: 'method-wrapper'
 # object has only read-only attributes
```

Listing 11: Function object and __call__ example

Finally, the scope object of a function in 12 is necessary as local variables inside a function should not be set as properties on the object of the function:

```
def a():
        x = 42
a.x # AttributeError
```

Listing 12: Function object and __call__ example

$$\begin{aligned} CallGraph &= (Node \times Node) \\ Analysis &= (Node \to State) \times CallGraph \end{aligned}$$

# 7 Results

# 8 Further work

andre magic methods

# Bibliography

[1] The Python Language Reference, Datamodel http://docs.python.org/2/reference/datamodel.html, May 13, 2013

[2] The Python Language Reference, Constants http://docs.python.org/2/library/constants.html, May 13, 2013

[3] The Python Language Reference, Built-in Types http://docs.python.org/2/library/stdtypes.html, May 13, 2013

[4] The Python Language Reference, __main__ http://docs.python.org/2/library/__main__.html, May 13, 2013

# Complete node reference

| | |
|---|---|
| FunctionDeclNode | `def f(...):` |
| ClassDeclNode | `class c(...):` |
| ClassEntryNode | |
| FunctionEntryNode | |
| ExitNode | |
| ConstantBooleanNode | |
| ConstantIntNode | |
| ConstantFloatNode | |
| ConstantLongNode | |
| ConstantComplexNode | |
| ConstantStringNode | |
| ConstantNoneNode | `None` |
| ReadVariableNode | `x` |
| ReadPropertyNode | $reg_{obj}$`.property` |
| ReadIndexableNode | $reg_{obj}$`[`$reg_{index}$`]` |
| WriteVariableNode | `x = value` |
| WritePropertyNode | $reg_{obj}$`.property = value` |
| WriteIndexableNode | $reg_{obj}$`[`$reg_{index}$`] = value` |
| DelVariableNode | `del x` |
| DelPropertyNode | `del` $reg_{obj}$`.property` |
| DelIndexableNode | `del` $reg_{obj}$`[`$reg_{index}$`]` |
| NoOpNode | |
| CallNode | $reg_{function}$`(`$reg_1$`, ..., `$reg_n$`)` |
| ReturnNode | `return` $reg_{value}$ |
| CompareOpNode | $reg_{left}$ $op_{comp}$ $reg_{right}$ |
| BinOpNode | $reg_{left}$ $op_{binop}$ $reg_{right}$ |
| IfNode | `if` $reg_{cond}$`, while` $reg_{cond}$`:` |
| RaiseNode | `raise` $reg_{exception}$ |
| ExceptNode | `except (`$type_1$`, ..., `$type_n$`):` |