Type Analysis of Python

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. Dynamic features of 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 figure 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)*.

```
 1   class Person(object):
 2     def __init__(self, name):
 3       self.name = name
 4
 5   class Student(Person):
 6     pass
 7
 8   s1 = Student('Foo')
 9   s2 = Student('Bar')
10
11   def addGrade(self, course, grade):
12     self.grades[course] = grade
13
14   Student.addGrade = addGrade
15
16   s1.grades = {}
18   s1.addGrade('math', 10)
19   s2.addGrade('math', 7)
```

Fig. 1

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 })*.

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 figure 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']*.

```
class Student(Person):
  def __getitem__(self, name):
    return self.grades[name]
  def __setitem__(self, name, val):
    setattr(self, name, val)
  def __getattr__(self, name):
    if name in self.grades:
      return self.grades[name]
    else:
      return "<no such grade>"
```

Fig. 2

Other

- Global variables are read only unless explicitly declared global.
- Multiple inheritance: Different lookup strategies depending on old-style (that does not extend *object*) or new-style classes (that extends *object*).

3. 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*).

According to the Python Language Reference the syntax of for loops is as follows:

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

An example could be:

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

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 element until the iteration is done and finally raises a *StopIteration* exception. Figure 3 below shows the generated control flow graph for the above program.

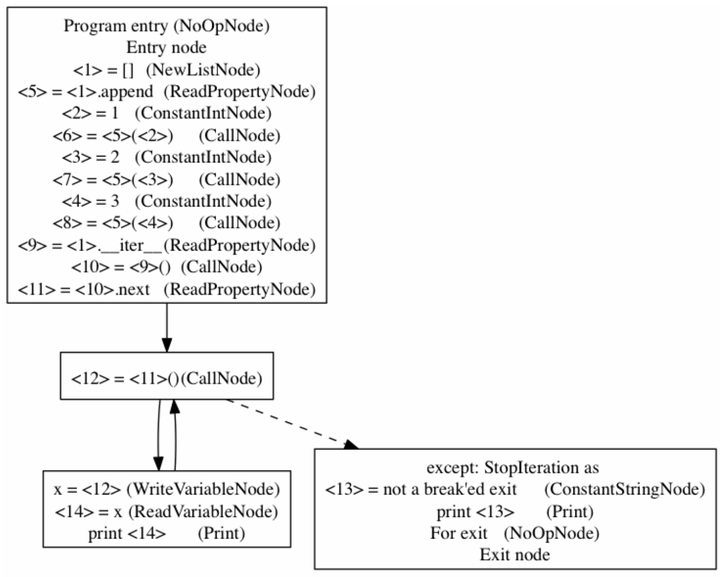
Program entry (NoOpNode)
Entry node
<1> = [] (NewListNode)
<5> = <1>.append (ReadPropertyNode)
<2> = 1 (ConstantIntNode)
<6> = <5>(<2>) (CallNode)
<3> = 2 (ConstantIntNode)
<7> = <5>(<3>) (CallNode)
<4> = 3 (ConstantIntNode)
<8> = <5>(<4>) (CallNode)
<9> = <1>.__iter__(ReadPropertyNode)
<10> = <9>() (CallNode)
<11> = <10>.next (ReadPropertyNode)
<12> = <11>()(CallNode)
x = <12> (WriteVariableNode)
<14> = x (ReadVariableNode)
print <14> (Print)
except: StopIteration as
<13> = not a break'ed exit (ConstantStringNode)
print <13> (Print)
For exit (NoOpNode)
Exit node

Fig. 3: The control flow graph of a simple for loop.

For a while loop we generate the control flow graph in figure 4.

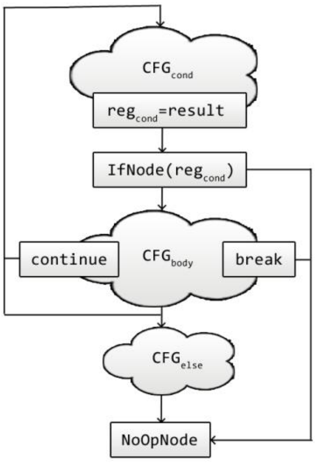

Fig. 4: The control flow graph of a while loop.

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 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.

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 (where the numbers to the left represent the result registers of the nodes).

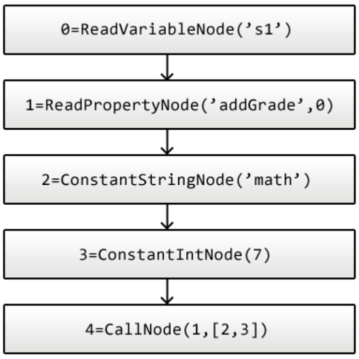

Fig. 5

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 because 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).

3.1 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.

For instance the code in the left column below is transformed into the code in the right column:

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

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

Fig. 6: A try-except-else-finally block before and after normalization, respectively.

Inductively, CFG's for the statement lists *`<try-stms>`* (CFG_{try}), *`<except-foo-stms>`* ($\text{CFG}_{\text{except-foo}}$), *`<except-stms>`* ($\text{CFG}_{\text{except}}$), *`<else-stms>`* ($\text{CFG}_{\text{else}}$), and *`<finally-stms>`* are created.

The CFG for the finally block is then cloned into three duplicates ($\text{CFG}_{\text{finally-normal}}$, $\text{CFG}_{\text{finally-handled-exc}}$, $\text{CFG}_{\text{finally-unhandled-exc}}$). 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. $\text{CFG}_{\text{finally-unhandled-exc}}$, 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.

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 $\text{CFG}_{\text{except-foo}}$. 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 (specifially $\text{CFG}_{\text{finally-unhandled-exc}}$). 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.

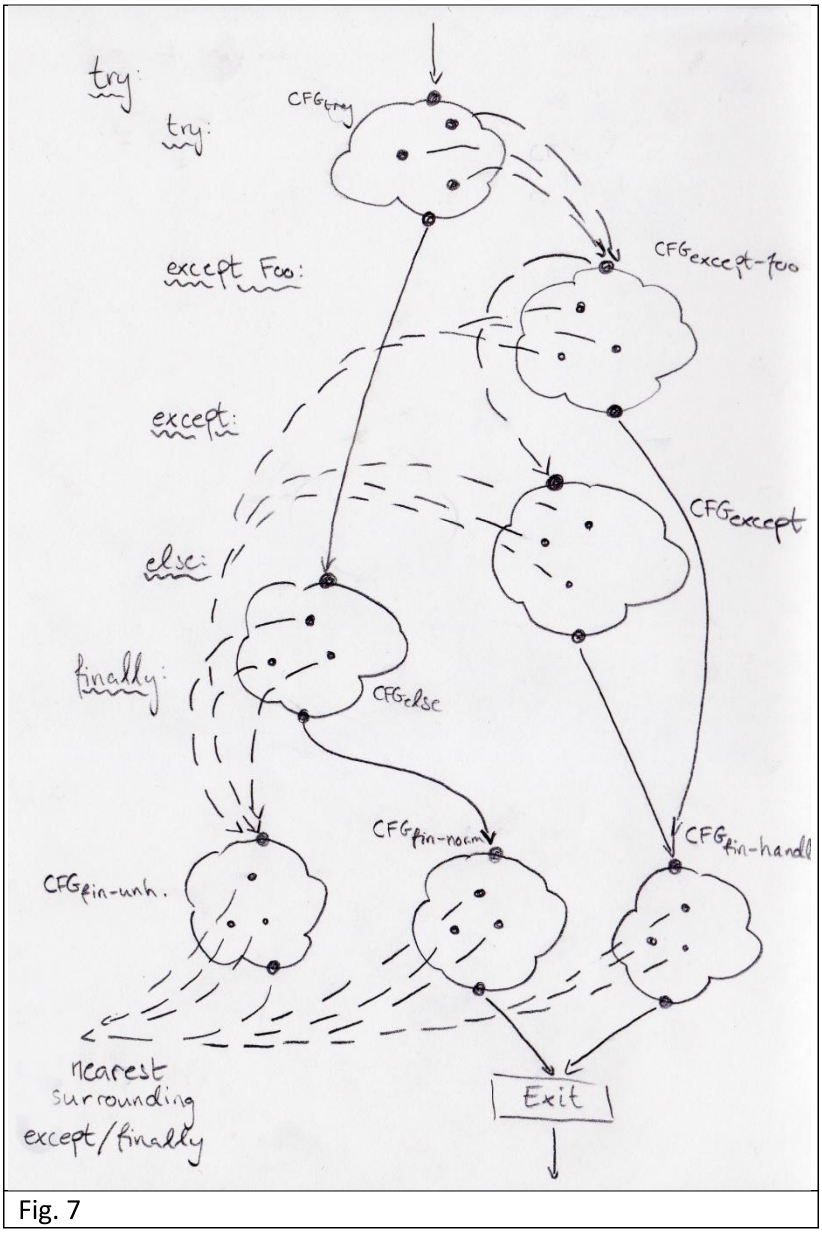
try:
try:
CFG_{try}
except Foo:
$\text{CFG}_{\text{except-foo}}$
except:
$\text{CFG}_{\text{except}}$
else:
finally:
CFG_{else}
$\text{CFG}_{\text{fin-unh.}}$
$\text{CFG}_{\text{fin-norm}}$
$\text{CFG}_{\text{fin-handl}}$
nearest surrounding except/finally
Exit

Fig. 7

The except blocks

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 $\text{CFG}_{\text{except-foo}}$ to the entry node of $\text{CFG}_{\text{except}}$.

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 $\text{CFG}_{\text{finally-unhandled-exc}}$). 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, $\text{CFG}_{\text{finally-handled-exc}}$).

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, $\text{CFG}_{\text{finally-unhandled-exc}}$).

In case the evaluation of the statements in the else block does not raise any exceptions, 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 $\text{CFG}_{\text{finally-normal}}$.

Complete node reference

FunctionDeclNode	def f(...):
ClassDeclNode	class c(...):
ClassEntryNode	
FunctionEntryNode	
ExitNode	
ConstantBooleanNode	
ConstantIntNode	
ConstantFloatNode	
ConstantLongNode	
ConstantComplexNode	
ConstantStringNode	
ConstantNoneNode	None
NewListNode	[]
NewDictionaryNode	{}
NewTupleNode	()
NewSetNode	set()
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 ($type_1$, ..., $type_n$):