Type Analysis of Python

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.

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'`.

```
 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

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 })`.

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']*.

```
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>"
```

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*).

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*). As a consequence, we generate the control flow graph illustrated in figure 3 for the while fragment.

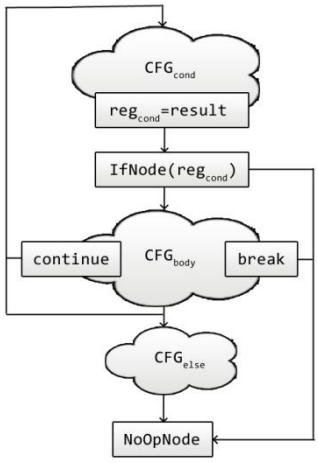

Fig. 3

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

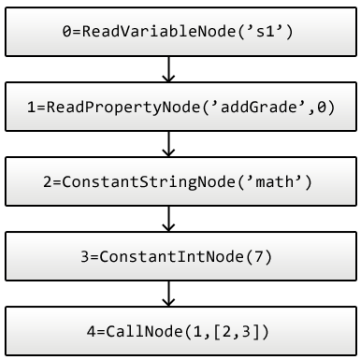

Fig. 4

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).

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}:
ForInNode	for ... in ...:
RaiseNode	raise $reg_{exception}$
ExceptNode	except ($type_1$, ..., $type_n$):

Code examples

Figure 1

```
class Person(object):
  def __init__(self, name):
    self.name = name

class Student(Person):
  pass

s1 = Student('Foo')
s2 = Student('Bar')

def addGrade(self, course, grade):
  self.grades[course] = grade

Student.addGrade = addGrade

s1.grades = {}
s1.addGrade('math', 10)
s2.addGrade('math', 7)
```

Figure 2

```
class Person(object):
  def __init__(self, name):
    self.name = name

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>"

s1 = Student('Foo')
s2 = Student('Bar')

def addGrade(self, course, grade):
  self.grades[course] = grade

Student.addGrade = addGrade

s1.grades = {}
s1.addGrade('math', 10)
```