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.

Abstract Values

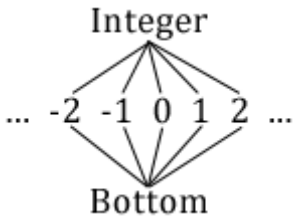
The concrete lattices follows below.

$$Value = Undefined \times None \times Boolean \times Integer \times Float \times Long \times Complex \times String \times P(ObjectLabel)$$

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*[1] which is returned, e.g.:

```
def a():
  pass
a() is None // true
```

Contrary to JavaScript, Python supports integers, floats, longs and complex numbers, so we have separate lattices for those. Note that *Complex = Float × Float*, since a complex number in Python is represented using a float for the real and imaginary part, respectively.[2] The *Integer* lattice is defined as follows:



The *Float*, *Long* and *String* lattices are defined in similar ways.

<Why we have chosen to model integers etc. as values when they are objects in Python>

Finally, a value can of course also be a pointer to an object in the heap, which we model in the *Value* lattice by having a power set[3] of object labels, *P(ObjectLabel)*.

Abstract State

We use the following lattice to model abstract state:

$$State = Heap \times Stack$$

[1] See http://docs.python.org/2/library/constants.html
[2] See http://docs.python.org/2/library/stdtypes.html#typesnumeric
[3] All our power sets are ordered by subset inclusion.

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

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

<TODO: Describe the Object lattice, and give examples of what the purpose of the power set scope chain is>

The Heap

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

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 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:

```
x = 10
def a():
  print x
```

For this particular program 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.

[4] See http://docs.python.org/2/library/__main__.html

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 (2) 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:

```
def a():
  pass
a.prop = 42
```

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"
```

Thus the function object in (3) 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
```

Finally, the scope object of a function in (4) 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
```

$$CallGraph = (Node \times Node)$$

$$Analysis = (Node \rightarrow State) \times CallGraph$$