A FORMAL SEMANTICS OF PYTHON 3.3

BY

DWIGHT GUTH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Associate Professor Grigore Roşu

Abstract

This thesis demonstrates the ability to formalize the operational semantics of complex programming languages in the $\mathbb{K}$ Semantic Framework, which provides an interpreter as well as analysis tools for exploring the state space of programs and performing static reasoning about programs. This is demonstrated by means of a partial semantics for the latest version of the popular Python programming language. With additional effort, this semantics will allow users to reason about Python programs, including sources of nondeterminism in the Python language specification, and formal reasoning about their behavior. While the semantics is incomplete, it is executable and has been thoroughly tested against a number of unit tests, and will be demonstrated to perform as well as the reference implementation of Python, CPython, on those features which have been completed. On these features, it also performs as well as or better than other comparable operational semantics of Python.

Table of Contents

Chapter 1

Introduction

This thesis demonstrates the ability to formalize the operational semantics of complex programming languages in the $\mathbb{K}$ Semantic Framework, which provides an interpreter as well as analysis tools for exploring the state space of programs and performing static reasoning about programs. This is demonstrated by means of a partial semantics for the latest version of the popular Python programming language. The $\mathbb{K}$ Framework then allows the construction of a number of useful tools from this definition.

In this chapter, we explain the context the semantics was created for, and our particular contributions. Chapter 2 provides the information necessary to comprehend the rest of the thesis, by describing the Python programming language and the $\mathbb{K}$ Semantic Framework, as well as briefly explaining the state of research into formal semantics. Chapter 3 provides a detailed summary of the semantics itself, by describing its organizational structure and its major components. Chapter 4 compares the semantics in detail with other operational semantics of Python, by testing it and them against each other and against the reference CPython implementation. Chapter 5 ends by summarizing our work and the semantics, as well as detailing future research opportunities for the semantics.

1.1 Problem Description

Research into programming language semantics and program analysis has been ongoing for a long time. Over the years, many formalisms have been proposed for expressing programming language semantics, each of which used by some researchers and not others, with no one style prevailing over all. However, despite all of these formalisms, most real programming languages used in industry are not designed or analyzed using formal semantics at all. Those analysis tools that do exist typically operate on simplified subsets of

programming languages, and most languages have not been formally defined in their complete depth at all.

The benefits of a semantics-based approach to software analysis are many, and include:

- A single testable semantics ensures that the behavior of the semantics can be trusted to soundly parallel reference implementations of the language;
- Detailing the semantics over and over in a multitude of different analysis tools is a waste of both time and effort;
- Basing analysis tools of of a formal semantics ensures that they are sound according to their stated purpose for program analysis.

Arguably, however, it is the lack of mainstream recognition of a single powerful formalism for capturing the semantics of languages that prevents this approach from being widely used. One argument for why no single formalism prevails is that many currently existing formalisms suffer from extensive weaknesses which prevent them from being scalable to large languages [21]. It is to address this lack that the $\mathbb{K}$ Framework [4, 19] was created. The $\mathbb{K}$ Framework is a rewriting-based formalism for defining programming languages, type systems, and calculi. $\mathbb{K}$ has demonstrated its feasibility by providing semantics to a number of academic languages [5, 9, 21, 22], subsets of several real programming languages [1, 7, 11, 12, 20], and, as a crowning achievement, the complete formal semantics of the C programming language [6]. $\mathbb{K}$ has a number of advantages over other semantic formalisms, including the ability to specify language rules modularly and to represent truly concurrent programming languages faithfully [21]. Also, with the complete semantics of C defined in $\mathbb{K}$, it is clear that $\mathbb{K}$ is able to scale up to very large programming language semantics.

However, not all programming languages are created equally. While C is indeed a quite large programming language, with a number of powerful static features and a depth of complexity provided by its notions of undefinedness in its standard, it has a very small standard library and provides very little in the way of native APIs, relying on its ability to link with native code to build up its functionality by means of an extensive system of library and include files that are not part of the language specification itself. It is also a very

low-level language, exposing memory to users and providing very little in the way of resource management or error-checking. This leads to a semantics which is very complex in certain ways and very simple in others.

We provide the formal semantics of Python to assuage the fears that the success of defining C in $\mathbb{K}$ is a fluke. Python 3.3 (the version we define) is a very high-level object-oriented language. It exposes only objects to the end-user, dynamic to the point that even the class of every object is itself an object. While the Python language specification does not explicitly expose a garbage collection algorithm, it provides hooks that any garbage collection interface must satisfy in order to meet the requirements of the language specification. It also has a very rich standard library defined both in pure Python code, and by means of a number of built-in modules, objects, and functions that the CPython reference implementation defines in C. Some of these pieces of functionality expose quite complex native APIs to the user, such as Python's `os` modules (see Section 3.2.14). It also has a number of quite complex reflective capabilities, including dynamic execution of code with the `compile`, `eval`, and `exec` functions (see Section 3.2.15). All of these combine to mean that the task of providing a complete semantics to Python requires a formalism capable of a number of advanced features for program state manipulation.

Because of this, despite a number of attempts, no complete formal semantics of Python has ever been made. Due to limitations of the scope of this thesis, the semantics we provide is not complete either. However, we argue that it is more complete by the nature of its construction than any formal semantics of Python that has preceded it (see Chapter 4). This definition in $\mathbb{K}$, constructed over the past two years of individual effort, is the largest $\mathbb{K}$ definition ever created to date and represents proof that $\mathbb{K}$ is capable of defining high-level, object-oriented, dynamic languages with complex standard libraries, as well as more straightforward, low-level languages like C. We discuss this semantics in detail in Chapter 3.

1.2 Contributions

The specific contributions of this thesis include:

- a detailed comparison of our semantics to other Python formalizations;

- the most in-depth formal semantics of Python to date, which has been thoroughly tested against a large library of unit tests;
- constructive evidence of that rewriting-based semantics scale;
- several features of the $\mathbb{K}$ framework which allow definitions to perform advanced program state manipulations such as I/O and dynamic code execution.

The tool, the semantics, and the test suite can all be found at `http://code.google.com/p/k-python-semantics/`.

1.3 Features

Our semantics captures as much of the semantics of the Python language specification as time allowed. We include below a partial list of features. Details concerning functionality outside the scope of the semantics can be found in Section 5.1. All of the features listed are given a direct and complete semantics. Where the phrase 'partial' is used, we refer only to a feature of which some subfeatures have been implemented and others have not, rather than to refer to a feature which is implemented in some simplified form.

- Expressions: Comprehensions, Displays, Literals, (partial) Yield Expressions, Environment Lookup, Primaries (function call, attribute reference, subscription), Arithmetic Operators, Boolean Operators, `lambda`;
- Statements: Expression Statements, Assignment Statements (including Augmented Assignment), Import Statements, Decorators, `assert`, `pass`, `del`, `return`, `raise`, `break`, `continue`, `global`, `nonlocal`, `if`, `while`, `for`, `try`, `with`, `def`, `class`;
- (partial) Built-in Functions: `abs`, `all`, `any`, `callable`, `classmethod`, (partial) `compile`, `eval`, `exec`, `format`, `getattr`, `globals`, `hasattr`, `hash`, `isinstance`, `issubclass`, (partial) `iter`, `len`, `locals`, `map`, `next`, `ord`, `range`, `repr`, `reversed`, `setattr`, `slice`, `classmethod`, `super`, `__import__`, `operator.index`;

- (partial) Built-in Types: (partial) `bool`, (partial) `bytearray`, (partial) `bytes`, (partial) `dict`, (partial) `float`, (partial) `int`, (partial) `list`, (partial) `object`, (partial) `set`, (partial) `str`, (partial) `tuple`, (partial) `type`, (partial) `types.ModuleType`, (partial) `types.CodeType`, (partial) `types.FrameType`, (partial) `types.FunctionType`, (partial) `types.GeneratorType`, (partial) `types.MethodType`, (partial) `types.TracebackType`;
- (partial) Built-in Exceptions;
- (partial) Built-in Modules: (partial) `_imp`, (partial) `_io`, (partial) `_weakref`, (partial) `errno`, (partial) `gc`, (partial) `posix`, (partial) `sys`.

Chapter 2

Background

In this chapter we provide details concerning the Python language specification and the CPython reference implementation, including some important definitions. We also explain the $\mathbb{K}$ Semantic Framework in which we give our semantics of Python.

2.1 Python Language Specification

The Python programming language was created in the early 1990s by Guido van Rossum. After the eventual creation of the Python Software Foundation in 2001, Python has gained in popularity and undergone several versions [15]. According to 2010 usage information for open source projects on Google Code [26], Python is the 7th most popular programming language, and it also ranks 8th in the June 2013 TIOBE index of programming language popularity [25], where it has been for over a year.

Python is primarily defined by means of a reference implementation maintained by the Python Software Foundation itself, called CPython. The latest version of CPython is version 3.3.2, released May 15th, 2013. However, several other versions of Python of note exist. PyPy [14] is a fast implementation of Python which uses JIT compilation. Jython [3] provides JVM support for Python, and IronPython [10] is an implementation of Python in the .NET framework maintained by Microsoft.

However, notably, none of these implementations of Python support version 3.0 or later of the Python language. Version 3.0 provided a number of changes to the language which were not backwards-compatible, and use of the version among industry is still not widespread. In the hopes of encouraging its use, the semantics we detail is a semantics of version 3.3, the latest version of the language.

In the construction of the semantics, we have had to face a number of diffi-

culties in deciding precisely where the boundaries lie between the Python language itself and the CPython implementation. The issue is less simple than it seems because while the online documentation claims to describe the Python language and to detail where certain features are implementation-specific, it suffers from a number of failings which make such a determination much more difficult. Any number of details of built-in functions and programming language features are described in vague terms so as to completely fail to mention a significant fraction of their actual behavior. Worse, some features are described in very concrete terms by the online documentation that are nonetheless factually incorrect when checked against the reference implementation. Still other features are not hinted at at all in the documentation and must be deduced from the behavior of the interpreter itself, or by examining its source code. Finally, it makes the determination that certain features of the Python language are CPython-specific, despite these same features having fundamentally irremovable effects on the behaviors of well-defined Python programs. We do our best to separate out all of these concerns and declare where a particular feature is implementation-specific, but the task is complex and incomplete in the work we present here.

However, some structure can be gleaned from the reference interpreter and the online documentation. The Python language, then, consists of a syntax of expressions and statements, a semantics for expressions which evaluate to objects, a semantics of statements which are executed for side effects, and a semantics for built-in modules, objects, and functions which are implemented by the implementation itself rather than in pure Python; many of which functions are hooked into the semantics of expressions and statements themselves.

2.2 Rewriting Logic and $\mathbb{K}$

The semantics of Python we provide is specified using a rewriting-based semantic framework called $\mathbb{K}$ [19]. In particular, the semantics is written using the $\mathbb{K}$ framework tool [4, 22], version 3.2. This version relies on the SDF (Syntax Definition Formalism) language [8] to generate a parser used to parse programs and definitions, a version of the `kompile` and `krun` tools written in Java, and a Maude [2] rewriting-logic engine to execute programs.

The rewriting performed by the $\mathbb{K}$ framework tool is based on Reachability Logic [18], a language-independent proof system for program verification in which rules of an operational semantics are axioms used to derive the steps of a computation. Rules contain patterns, which are satisfied if they *match* a configuration. Essentially, given some term allowed by the signature of the semantics (i.e., a program together with its input), we deduce the output of the term in the semantics by means of repeatedly applying rules in the operational semantics. This yields a transition system for any program. A single path of rewrites describes the behavior of an interpreter, whereas multiple paths through the transition system represent all behaviors of a nondeterministic program.

In this thesis, $\mathbb{K}$ can be treated as a front-end to a subset of Reachability Logic, designed to present those features needed to describe programming languages as compactly as possible. In $\mathbb{K}$, a program's state is described using a nested set of containers called *cells*, as seen in Figure 2.1. These collections contain pieces of the program state, such as a computation stack or continuation (k), stacks (e.g., cstack), or a heap (store), etc.

As an example to understand the types of rules present in the semantics, consider a typical rule for a simple imperative language (see Section 3.2.9 for the much more complicated version in Python) for adding two objects:

```
rule <k> O:Object + O2:Object => O . __add__ (O2) ...</k>
```

We see here only one cell: k. The k cell represents a list of computations to be performed, the leftmost of which is the next computation to be performed. The rule above says that if the next thing to be evaluated is one object being added to another object, replace it with a call to the __add__ function.

This exhibits a few basic features of $\mathbb{K}$. First, rules only mention the cells they need in the rule. If we needed another cell, we could mention it using the same syntax as the above rule and Figure 2.1. Second, we can omit part of a cell using "...". For example, in the above k cell, we are not interested in what is below the current term on the stack. Finally, we represent rewrite operations with the => operator. In the above example, we replace the current term, but not whatever is beneath it on the stack. By means of this local rewrite operator, we save space. For example, the same rule written as a traditional rewrite rule would be:

```
rule <k> O:Object + O2:Object ~> K:K </k> =>
```

```
<k> O . __add__(O2) ~> K:K </k>
```

This small change becomes quite significant in more complex rules, allowing us to prevent duplicating large portions of every rule, which would need to be changed in multiple places every time the semantics is updated.

Note that the above rule uses a new operator, `~>`. This operator is used to indicate that one computation follows another. Consider the `pass` statement in Python:

```
rule pass => .K
```

This rule omits referencing any cell at all. As such, it is equivalent to the following rule:

```
rule <k> pass => .K ...</k>
```

The `.K` in this rule corresponds to the unit of computation. Thus if the `k` cell contained `pass ~> pass`, the above rule would rewrite it to `.K ~> pass` which is equivalent to `pass`. As you see, this allows the `~>` operator to construct a list that is used to represent a sequence of computations.

We also use the `_` symbol to represent an anonymous unnamed variable. Finally, in order to get the appropriate terms to the top of the `k` cell (i.e., to decide what term to rewrite next), the grammar of Python is annotated with "strictness" annotations. For example, we declare the above-used addition operation using:

```
syntax Exp ::= Exp "+" Exp [seqstrict]
```

This means that the operands of the addition expression will be evaluated sequentially from left to right. In contrast, the `if` construct looks like this:

```
syntax Stmt ::= "if" Exp ":" Stmts "else" ":" Stmts [strict(1)]
```

indicating that only the first argument can be lifted for evaluation. The two annotations above automatically generate the following six rules, known as heating and cooling rules:

```
configuration <T>
                <k> initialize($PGM:K) </k>
                <control>
                  <currentFrame>
                    <frameObject> .K </frameObject>
                    <xstack> .List </xstack>
                    <xcontext> ref("None") </xcontext>
                    <lstack> .List </lstack>
                    <fstack> .List </fstack>
                  </currentFrame>
                  <cstack> .List </cstack>
                </control>
                <scope> .List </scope>
                <store>
                  <object multiplicity="*">
                    <id> 0 </id>
                    <oenv multiplicity="?"> .Map </oenv>
                    <oattrs> .Map </oattrs>
                  </object>
                </store>
                <gc> 1 </gc>
                <gcThreshold> $GCTHRESHOLD:Int </gcThreshold>
                <literals> .Map </literals>
                <symbols> .Map </symbols>
                <builtinObjectsWithNewMethod>
                  .Set
                </builtinObjectsWithNewMethod>
                <builtinModules> .Map </builtinModules>
                <references> .Map </references>
                <bootstrapping> 3 </bootstrapping>
                <nextLoc> 1 </nextLoc>
                <sysPath> $SYSPATH:List </sysPath>
                <sysArgv> $SYSARGV:List </sysArgv>
                <environ> $ENVIRON:Map </environ>
                <constants> $CONSTANTS:Map </constants>
                <importlib> $IMPORTLIB:K </importlib>
                <optimize>
                  $CONSTANTS:Map ( "PYTHONOPTIMIZE" )
                </optimize>
              </T>
```

Figure 2.1: Python Configuration

```
rule E1 + E2 => E1 ~> HOLE + E2 [heat]
rule V1 ~> HOLE + E2 => V1 + E2 [cool]
rule V1:KResult + E2 => E2 ~> V1 + HOLE [heat]
rule V2 ~> V1 + HOLE => V1 + V2 [cool]
rule if E1: S1 else: S2 => E1 ~> if HOLE: S1 else: S2 [heat]
rule V1 ~> if HOLE: S1 else: S2 => if V1: S1 else: S2 [cool]
```

Here, `E1` and `E2` are unevaluated expressions, whereas `V1` and `V2` are evaluted expressions (i.e., values). By convention, we declare that a particular term is a value by making it part of the syntactic sort `KResult`. `V1:KResult` therefore is a variable of sort `KResult`, i.e., a value. Thus the second operand of `+` only evaluates if the first has already been evaluated. Note that as written, the heating and cooling rules can be applied indefinitely. This is deliberate, to allow the full generality of nondeterminism to be expressed. In practice, most operands will only heat if they are not values, and only cool if they are values. More information on these annotations can be found in documentation on $\mathbb{K}$ [19].

For the remainder of the thesis, we use the following conventions with respect to the types of variables: `O` is of sort `Object`. `E` is of sort `Exp`. `S` is of sort `Stmt` or `String` or `Set`. `Ss` is of sort `Stmts`. `X` is of sort `Id`. `I` is of sort `Int`. `B` is of sort `ObjId`. `C` is of sort `Configuration` (i.e., `Bag`). `M` is of sort `Map`. `L` is of sort List.

At this point, we have only looked at the most basic features of $\mathbb{K}$, but they should be enough to understand the semantics in the thesis. People interested in learning the rest of the $\mathbb{K}$ language are encouraged to take the $\mathbb{K}$ tutorial [17].

Chapter 3

Semantics of Python

This chapter describes the structure and major components of the $\mathbb{K}$ semantics of Python 3.3. We describe each of the major features of the Python language as well as provide snippets of their implementation, as well as discussing several advanced features of the $\mathbb{K}$ framework used to implement these components.

3.1 Introduction

In this chapter, we present the major components our executable, testable formal semantics. Formally speaking, our semantics implements version 3.3.2 final of the Python programming language. In practice, it implements only a subset of the features of that version, however. The ultimate goal of providing this semantics, once it is eventually completed, is to provide practical tools for reasoning with and manipulating Python programs. In practice, it is not in a state that is ready to perform more tasks than the execution of programs.

The semantics defines 45 modules, 1406 sentences (mainly rules), 608 productions, and 32 cells in its configuration, defined over 4611 lines of code. It implements 246 built-in functions. Of the rules, roughly 40% are associated with assigning semantics to components of Python syntax, roughly 40% are associated with assigning semantics to built-in functions, and the remaining 20% are associated with the organizational structure of the semantics.

3.2 The Semantics of Python in $\mathbb{K}$

In this section, we describe the different components of the semantics and provide a number of example rules from the semantics. These rules may be modified slightly from their original form. The complete semantics with the

original form of the rules can be found online at `http://code.google.com/p/k-python-semantics/`, though the rules there are less well-explained.

3.2.1 Syntax

The parser used for our programs is written in Python, using the `ast` module that Python provides. While Python's `ast` module performs a number of simplification steps to make its internal compiler easier to implement, we attempt to preserve the original syntax of the language when translating into $\mathbb{K}$ whenever possible. Certain simplification steps are performed because we operate on an AST instead of a parse tree, but Python's grammar is notoriously convoluted and requires a number of very advanced features which would make writing the semantics itself in the same grammar very difficult, even if it could be done. Future work intends to implement a complete parser for Python in $\mathbb{K}$, however.

3.2.2 Configuration

The configuration of a Python program contains 31 distinct cells, as shown in Figure 2.1. The outermost cell T wraps the entire configuration so it can be manipulated as a unit if necessary. The k cell, as mentioned before, stores the stack of computation steps. Inside the control cell is the cstack cell which stores the call stack of functions invoked and returned. It also contains the currentFrame cell which contains the data to be saved and restored with each function call. This includes the frame object (frameObject), the stack of exceptions (xstack), the current exception context (xcontext), the loop stack (lstack), and the finally stack (fstack).

At the top again, the scope cell is used to store information about lexically nested scopes for use computing environment lookups. The store cell contains a set of object cells, each with an identifier (id), an optional environment (oenv), and a set of attributes (oattrs). The gc and gcThreshold cells are used to control the frequency of garbage collection. The literals, symbols, builtinObjectsWithNewMethod, and builtinModules cells are used to cache objects for global lookup. The references and bootstrapping cells store information needed to bootstrap the Python interpreter into functioning. The nextLoc cell is used to obtain unique ids for objects. The sysPath, sysArgv, environ

and constants cells store metadata used to tweak the function of the Python interpreter, such as `sys.path`, `sys.argv`, `os.environ`, `sys.hash_info`, etc. The importlib cell stores the Python code used to bootstrap the `__import__` function, and the optimize cell stores the current value of the optimize flag, which controls `assert` statements and docstrings.

3.2.3 Common Basic Operations

In order to modularize the semantics, a number of auxiliary operations are defined by the semantics to be used by the other rules. In order to understand the rules that follow, a few of the more basic of these must be discussed and explained.

The first and simplest of these is the construct for objects:

```
syntax Object ::= obj(ObjId, Bag)
syntax ObjId ::= Int | String
```

An object has an identifier and a set of attributes. The identifier can be an integer, as with most objects, or a string, in the case of built-in functinos. The attributes contain information concerning the object itself, such as are stored by CPython in *slots*. For example, the object `None` is represented by the following term:

```
obj(24, <oattrs>
          "__class__" |-> ref("NoneType")
          "__name__" |-> "None"
        </oattrs>)
```

This shows that the class of `None` is the NoneType.

The second major auxiliary operator can already be seen above:

```
syntax Exp ::= ref(ObjId)
```

The `ref` operator represents a reference to an object, and, appropriately, evaluates to the object it is a reference to. Thus, `ref("None")` evaluates to the object displayed above, as does `ref(24)`. This is because a number of built-in objects must be referenced by the semantics, and therefore are given a string name instead of only being referred to by an integer. For example,

you will see in Figure 2.1 that the xcontext cell begins by default by containing a reference to the `None` object. In a number of places throughout this document, we refer to objects by name (AttributeError, TypeError, super, etc.) to refer to an absolute reference of this type. We do this to simplify the display of a number of rules that need to remain correct even when a user modifies the `builtins` module itself. Nonetheless, in the complete semantics these references are not identifier lookups, despite our borrowing that syntax for display purposes.

The next major auxiliary operator represents a chain of expressions:

```
syntax K ::= K "->" K [strict(1), right]
syntax Null ::= ".Obj"
syntax Exp ::= Null
rule O:Object -> _ => O
rule .Obj -> Chain => Chain
```

As you can see, we define a new expression here, `.Obj`. This serves a similar function to the C `NULL` in CPython, representing the absence of an object rather than any particular object. We use this instead of `None` because there are cases where `None` represents an object like any other, and an instance of the `None` object represents a different semantics than the absence of any object whatsoever. The arrow operator can be thought of as chaining together a number of attempts to obtain an object. If the first attempt succeeds, it returns that object. If instead the expression evaluates to `.Obj`, we attempt to evaluate the next expression in the chain. If none of the expressions succeed, the operator returns `.Obj`.

Next, we cover manipulation of object attributes, as seen in Figure 3.1.

`getattr` is a function that takes an object and a string and looks up an attribute that is not an object. `getref` is a function that does the same thing but looks up an object attribute. `getattr2` and `getref2` do the same but with strictness. `hasattr` returns true if the attribute exists and false otherwise. And `setattr` and `setref` perform the inverse function of set on attributes.

Finally, we discuss invocation of built-in functions:

```
syntax Exp ::= invokeBuiltin(Object, List, Map)
```

```
syntax K ::= getattr(Object, String) [function]
rule getattr(obj(_,<oattrs> Attrs:Map </oattrs>), S:String)
  => Attrs:Map(S)

syntax K ::= getattr2(Exp, String) [strict(1)]
rule getattr2(O:Object, S:String) => getattr(O, S)

syntax Exp ::= getref(Object, String) [function]
rule getref(obj(_,<oattrs> Attrs:Map </oattrs>), S:String)
  => Attrs:Map(S)

syntax Exp ::= getref2(Exp, String) [strict(1)]
rule getref2(O:Object, S:String) => getref(O, S)

syntax Bool ::= hasattr(Object, String) [function]
rule hasattr(obj(_,<oattrs> Attrs:Map </oattrs>), S:String)
  => S in keys Attrs

syntax K ::= setattr(ObjId, String, K)
rule <k> setattr(B, S, K) => . ...</k>
     <object>...
       <id>B</id>
       <oattrs> Attrs => Attrs[K / S] </oattrs>
     ...</object>

syntax K ::= setref(ObjId, String, Exp) [strict(3)]
rule <k> setref(B, S, O) => . ...</k>
     <object>...
       <id>B</id>
       <oattrs> Attrs => Attrs[O / S] </oattrs>
     ...</object>
```

Figure 3.1: Manipulation of Python Object Attributes

This piece of syntax is used to hook together the invocation of a built-in function with its implementation. Consider, for example, the trivial built-in function `int.__int__`:

```
rule invokeBuiltin(obj("int.__int__",_), ListItem(Self), .)
  => Self
```

This rule says that to invoke the built-in function in question, you need a single positional parameter, Self. The function then returns that parameter. A function with more parameters would have more ListItems, and a function with keyword arguments would have MapItems in the third argument.

3.2.4 Attribute Reference

Attribute reference in Python follows a number of distinct layers. At the top is the attribute reference syntax:

```
syntax NAME ::= Id
syntax TargetAndExp ::= Exp "." NAME [strict(1)]
rule O:Object . X => getattr(O, Id2String(X))
```

This says that an attribute reference is an expression followed by a dot and an Id. It then evaluates to an invocation of the `getattr` function with two arguments: the evaluated object being dereferenced, and the string form of the identifier.

Next, we show the implementation of the `getattr` function, as seen in Figure 3.2.

This says that if you invoke `getattr`, raise a TypeError if the second argument is not a string. Try to invoke the `__getattribute__` function on the object with the string as argument. If it succeeds, return that object. If it fails with an AttributeError, try the same with the `__getattr__` function. If that succeeds, return that object, otherwise, return the first AttributeError, unless the function is called with three arguments, in which case return the third argument.

Now, however, we must define the `__getattribute__` built-in functions (Figure 3.3).

This says that if you invoke the `__getattribute__` function on an instance, raise a TypeError if the second argument is not a string. Check to see if the

```
rule invokeBuiltin(obj("getattr",_),
                   ListItem(O)
                   ListItem(S),
                   .)
  => if isinstance(S, str):
       .K
     else:
       raise TypeError ~>
     try:
       getmember(O, "__getattribute__", true, false, true) (S)
     except AttributeError:
       (getmember(O, "__getattr__", true, false, false)
           -> raise) (S)
```

Figure 3.2: `getattr()` Implementation

```
rule invokeBuiltin(obj("object.__getattribute__",_),
                   ListItem(O)
                   ListItem(O2:Object),
                   .)
  => checkData(getmember(O, O2, false, false, false), O)
         -> mapLookup(O, O2)
         -> getmember(O, O2, true, false, true)

rule invokeBuiltin(obj("type.__getattribute__",_),
                   ListItem(O)
                   ListItem(O2),
                   .)
  => checkData(getmember(O, O2, false, false, false), O)
         -> getmember(O, O2, true, true, true)

rule checkData(O:Object, O2:Object)
  => if ((getmember(O, "__set__", false, false, false) ->
          getmember(O, "__delete__", false, false, false))
              ==Obj .Obj):
       .Obj
     else:
       descriptor(O, O2, gettype(O2), false)
rule checkData(.Obj, _) => .Obj
```

Figure 3.3: `__getattribute__()` Implementation

object obtained by performing the attribute reference without a descriptor is a data descriptor (i.e., has a `__set__` or `__delete__` attribute). If so, invoke the descriptor. Otherwise, look up the attribute in the dict of the object and return it if present. If it is not present, proceed to look up the attribute in the dict of the type of the object. Performing `__getattribute__` on a type object is the same, except if the lookup succeeds on the instance rather than the type of the object, the descriptor is invoked. We will not discuss the details of the implementation of `mapLookup`, `getmember`, or `descriptor` except to say a few words:

- Lookup on the type of an object performs a lookup on all the base classes of that type in method resolution order;
- Calling `getmember` skips invoking a descriptor if the third parameter is false;
- Calling `getmember` performs implicit lookup if the fourth parameter is false (see [16]);
- Calling `getmember` raises an AttributeError on failure if the fifth parameter is true;
- The descriptor of `A.x` is invoked with arguments `(None, A)` if the attribute is found in the dict of the instance, and with arguments `(A, type(A))` if it is found in the dict of the class or one of its base classes. Hereafter we shall refer to the first form of these arguments an an *instance-form descriptor call*, and the second as a *class-form descriptor call*;
- Contrary to the above, the descriptor of `None.x` is invoked with arguments `(None, None)`;
- Descriptor lookup does not invoke a descriptor itself.

3.2.5 Methods

A method in Python is a callable object that combines another callable object with an object that represents its first parameter (or, equivalently, the object the method belongs to). Methods are constructed when a descriptor of a function object is called in class-form, as seen in Figure 3.4.

```
rule invokeBuiltin(obj("builtin-method.__get__",_),
                   ListItem(Self:Object)
                   ListItem(Instance:Object)
                   ListItem(Owner:Object),
                   .)
  => describe(Self, Instance, Owner, "get_function")

rule describe(Self, Instance, Owner, S) =>
     doDescribe(Self, Instance, gettype(Instance), S)
when Owner is None orBool Instance is not None

rule doDescribe(Self, O:Object, _, "get_function")
  => method(Self, O)

rule method(Func, Self)
  => immutable("__func__" |-> Func
               "__self__" |-> Self,
               types.MethodType)
```

Figure 3.4: `__get__()` Implementation for Built-in Methods

This says that if you have a built-in method (i.e., a built-in function that should have a `__get__` descriptor), its get descriptor should, if called in instance-form, return itself, and, if called in class-form, return a method object whose `__func__` attribute is the built-in method and whose `__self__` attribute is the first argument passed to the descriptor. Some built-in fumnctions have a `__get__` descriptor and some do not. This is because sometimes they need to be called without a self parameter, (e.g., `int.__new__`), and sometimes with one (e.g., `int.__add__`).

```
rule invokeBuiltin(obj("method.__call__",_),
                   ListItem(obj(N, <oattrs>...
                                   "__func__" |-> Func
                                   "__self__" |-> Self
                                ...</oattrs>))
                   L:List,
                   M:Map)
  => doCall(Func, ListItem(Self) L, M)
```

This is the implementation of the semantics for calling a method object. The `doCall` auxiliary function constructs a function application AST node

with the given set of positional and keyword arguments. It then calls the underlying function with the first positional argument provided by the `__self__` attribute.

3.2.6 Code Objects and Scoping

One of the inherent difficulties in creating a semantics of Python is the question of scoping. Python supports free variables in functions being bound by declarations in an outer scope, as well as explicit declarations to control the scope of variables. CPython implements this by means of compiling a set of statements into a code object containing Python bytecode and a certain amount of metadata. We do not create bytecode in our semantics. However, in order to process code to generate the information needed to properly handle the scoping of variables, we do generate code objects. This is because execution of a function statement requires that actions be taken which are dependent on the scoping of variables in child scopes. Without code objects, it would be impossible to determine how to create the closures that are needed in order to invoke a function from outside the scope that created it.

The rules needed to generate a code object from its name, parameters, body, and type are quite complex and cannot easily be understood if printed here. Therefore, we choose to summarize instead the actions taken when a code object is constructed at compile time. Essentially, code objects have a number of attributes which provide information needed to be able to invoke them again later. `co_argcount`, for example, stores the number of positional arguments a function takes. Similarly, `co_kwonlyargcount` stores the number of keyword-only arguments. `co_flags` stores, among other things, whether a function takes a vararg or kwarg parameter. `co_code` is adopted from CPython, but instead of storing bytecode it stores a $\mathbb{K}$ term of the AST of the function. Finally, a number of fields are used to store the information necessary to determine how to scope a particular variable name in a function. `co_freevars` stores the variables that are free (i.e., not bound by that function but by an outer scope). `co_cellvars` stores the variables that are free in a child scope and bound by the current scope. `co_varnames` stores the local variables of the function. Finally, `co_names` stores the global variables in a function as well as the local variables of a class scope.

We also make use of several additional fields that are not part of CPython

in order to be able to assign variables to the environment without resorting to further AST modifications. `co_globals` is the list of variables explicitly declared global in a scope. `co_nonlocals` does the same with the `nonlocal` declaration. And `co_type` stores whether a scope is a class scope or a function/module scope.

In order to compute both of these last two sets of fields, it is necessary to traverse the AST of the body of a function and process it. This traversal performs several tasks. First, it performs optimization of the code by removing assertions and docstrings as needed based on the current optimization level. Second, it collects all definitions and uses of all variables in the function. Third, it collects all `nonlocal` and `global` declarations in the function. Fourth, it traverses all the nested scopes (i.e., generators, classes, nested functions, etc) inside the function and recursively creates code objects for them. Fifth, it performs a number of checks on the syntax of the function to determine whether a SyntaxError should be raised.

After it has done all of these things, it takes the information it has computed and assigns values to the tuples concerning scoping information it has determined. Because these rules are subtle and complex, we will provide and explain the rules concerning computation of scoping of variables in Figure 3.5.

Essentially what we are doing here is computing three sets. The first, LOCAL, is the set of variables defined in the current scope. The second, FREE, is the set of variables declared in the current scope or a parent scope which are used in a scope that is a child of the scope they are defined in. The third, BOUND, is the set of variables declared in a parent scope which, if used in the current scope, would be accessible. Thus we compute FREE based on BOUND by means of deciding which variables we use are defined in a parent scope and which variables we use are defined in the current scope and therefore local variables, despite being defined in a parent scope. In essence, `co_varnames` contains variables in LOCAL but not in FREE, `co_freevars` contains variables in FREE but not in LOCAL, `co_cellvars` contains variables in both FREE and LOCAL, and `co_names` contains variables in neither FREE nor LOCAL. This is made more complicated by the concerns of scoping variables in class scopes, but we will discuss that in Section 3.2.11.

```
rule LOCAL(Defs, Globals, Nonlocals, Params, Type)
  => ((Defs Setify(PARAMS(Params))) -Set Globals)
         -Set Nonlocals
when    Setify(PARAMS(Params)) &Set Nonlocals ==Set .
andBool Setify(PARAMS(Params)) &Set Globals ==Set .
andBool Type =/=K typeobject

rule FREE(Defs, Uses, Globals, Nonlocals, ChildFree, L)
  => Nonlocals ChildFree (((Uses) &Set BOUND(.,L))
         -Set Globals)
when    (Nonlocals &Set BOUND(., L)) ==Set Nonlocals
andBool notBool "super" in Uses

rule FREE(Defs, Uses, Globals, Nonlocals, ChildFree, L)
  => Nonlocals ChildFree (((Uses SetItem("__class__"))
       &Set BOUND(.,L)) -Set Globals)
when    (Nonlocals &Set BOUND(., L)) ==Set Nonlocals
andBool "super" in Uses

rule BOUND(Bound:Set, L ListItem(scope(_, Defs, _, Nonlocals,
         Globals, _, Params, funcobject, _)))
  => BOUND((Bound LOCAL(Defs, Globals, Nonlocals, Params,
         funcobject)) -Set Globals, L)

rule BOUND(Bound:Set, L ListItem(scope(_, _, _, _, Globals,
         _, _, moduleobject, _)))
  => BOUND(Bound, L)

rule BOUND(Bound, .) => Bound
```

Figure 3.5: Python Scoping Rules

3.2.7 Environment Lookup, Assignment, and Deletion

The rules pertaining to environment manipulation being quite verbose, we merely summarize their behavior in the form of an algorithm:

1. If an environment lookup is looking up a variable in the LOCAL set of the current frame, look it up in the `f_locals` dictionary. If it is not present there, raise an UnboundLocalError;

2. Otherwise, if the environment lookup is of a variable in the FREE set, look it up in the appropriate `cell` object for the current scope. If it is not present there, raise a NameError;

3. Otherwise, try looking it up in order in the `f_locals`, then `f_globals`, then `f_builtins` dictionary;

4. If it is not present in any of these, raise a NameError.

Similarly, we summarize the rules for environment assignment/deletion:

1. If the variable is in the FREE or LOCAL set of the current frame, perform item assignment/deletion on the `f_locals` dictionary. A deletion of an item not present raises an UnboundLocalError;

2. If the variable is in the FREE set of the current frame, perform item assignment/deletion on the appropriate `cell` object;

3. If the variable is FREE and not LOCAL, perform environment assignment/deletion again on the scope it is LOCAL in;

4. If the variable is in neither FREE nor LOCAL, perform item assignment/deletion on the `f_globals` dictionary. A deletion of an item not present raises an NameError.

Note that environment assignment/deletion of variables in a class scope obeys different rules because the scoping is different. We discuss these alternate rules in Section 3.2.11.

3.2.8 Function Calls

Once code objects have been created, there remains the task of generating the function objects themselves, and then invoking those objects. This is tied intricately in with scoping because of the way in which cells are used to propagate closures from an outer scope to an inner scope. In essence, there are four fields of note in a function object that are created at function declaration time and are used in order to invoke the function. `__defaults__` contains a tuple of the default values for positional arguments to the function. These defaults are loaded at invocation time if a particular position argument is not provided. `__kwdefaults__` does the same with the default values for keyword-only arguments. `__closure__` contains a tuple of cells used to store free variables. `__globals__` contains the dictionary of global variables for the module the function was declared in.

When a function is invoked, a number of steps are performed in order to fill in arguments into the parameters of a function. This is a somewhat complex procedure, and we will not go into the details of the rules that it requires, but instead merely summarize the behavior.

The first thing that happens is the function arguments are evaluated from left to right, and sorted into four categories: positional arguments, keyword arguments, a stararg, and a kwarg. Stararg arguments are unrolled using an iterator into positional arguments, and kwarg arguments are unrolled using a mapping into keyword arguments. Then the task of filling in arguments begins. The first thing that occurs is the positional arguments are put into positional parameters from left to right. If any are left over, they are stored in the vararg parameter if one exists, and a TypeError is raised otherwise. If there are not enough positional arguments, keyword arguments go to fill the positional arguments if they share a name. Otherwise, if a default positional argument is provided, it is filled in. If this is still not sufficient to fill the positional parameters, a TypeError is raised. A TypeError is also raised if a parameter is specified both positionally and as a keyword argument. Next, the remaining keyword arguments are used to fill in keyword-only parameters. If there are too many, a dict is created for them if kwargs are present, otherwise a TypeError is raised. If there are not enough, default values for keyword-only arguments are used. If this is still not enough, a TypeError is raised. After this point, matching arguments has completed

and we have a map that maps the names of parameters to their values.

```
rule <k>
       invoke(obj(_:Int, <oattrs>...
                           "__closure__" |-> Closure
                           "__code__" |-> Code
                           "__globals__" |-> Globals
                         ...</oattrs>), M) ~> K:K
       =>
       executeFrame(N, Code, ref(Frame),
           makeLocals(...),
           Globals, Globals ["__builtins__"],
           makeCells(...),
           M) ~> return
     </k>
     <nextLoc> N => N +Int 1 </nextLoc>
     <control>...
       <currentFrame>
         <frameObject> Frame:Int => N </frameObject>
         C:Bag
       </currentFrame>
       <cstack>
         . => ListItem(call(Frame, C, K))
       ...</cstack>
     ...</control>
```

The above abbreviated rule performs the invocation of the function itself. The closure and the globals are obtained from the function object. The local variable dictionary for the new frame is generated by filling in all the free variables from the closure. The namespace for built-ins is taken from the `__builtins__` tiem in the globals dictionary. Cells for free variable storage are created for every variable whose name is in `co_cellvars`, and cells are copied from one frame to the next for every variable whose name is in `co_freevars`. Then all the relevant information from the previous call frame is pushed into the stack and a fresh call frame is created.

```
rule return => return None [macro]
```

```
rule <k> return O ~> _ => O ~> K </k>
     <control>...
       <cstack>
         ListItem(call(Frame, C, K)) => .
       ...</cstack>
       <currentFrame>
         <frameObject> _ => Frame </frameObject>
         (_ => C)
       </currentFrame>
     ...</control>
```

Returning from a function occurs when a return statement is reached. A return without arguments returns the value `None`. Return occurs by popping everything off the stack again, removing the call frame, and then pushing the return value onto the `k` cell, where it replaces the function invocation node that was there before, and is cooled as the result of the function call.

3.2.9 Arithmetic and Boolean Operations

Compared to what has come before, arithmetic operations in Python are relatively straightforward. We will demonstrate the flow of operations by means of the example of addition, as seen in Figure 3.6.

Adding two objects occurs by attemping to find an `__add__` or `__radd__` function to invoke. If the right operand is an instance of a subclass of the left operand's type, we look first for an `__radd__` attribute on the right operand. Otherwise we search first for an `__add__` attribute on the left operand. We decide whether the operator is suitable if we find it by calling it and checking whether it returns `NotImplemented`. A method that returns this value is treated identically to if the method does not exist. If no suitable operator is found, a TypeError is returned.

Unary operators behave identically to binary operators except that there is no reflected method.

Boolean operators are slightly more complicated because they can be connected to form a non-associative list. Thus `None is None is None` evaluates to `True` while `(None is None) is None` and `None is (None is None)` evaluate to `False`. To deal with this, we parse a single multi-part comparison

```
eyntax Exp ::= Exp "+" Exp [seqstrict]

rule O + O2
  => coerceBinary(O, O2, "__add__", "__radd__", "+")

rule coerceBinary(O:Object, O2:Object, X, RX, S:String)
  => coerceBinaryBase(O, O2, X, RX) ->
         (raise TypeError)

rule coerceBinaryBase(O:Object, O2:Object, X, RX)
  => (if hasbase(getbases(gettype(O2)), gettype(O)):
        coercion(getmember(O2, RX, true, false, false) (O))
      else:
        .Obj) ->
      coercion(getmember(O, X, true, false, false) (O2)) ->
      coercion(getmember(O2, RX, true, false, false) (O))

rule coercion(O:Object) =>
     if O is NotImplemented:
       .Obj
     else:
       O

rule coercion(.Obj) => .Obj
```

Figure 3.6: Addition of Expressions

operation with a wrapper KLabel. We will demonstrate with == and != but the technique generalizes to all comparison operators.

```
syntax Exp ::= Compare [klabel('Compare), strict]

syntax Compare ::= right:
                     Exp "==" Compare       [compare]
                   | Exp "!=" Compare       [compare]
                   | Exp "==" Exp       [compare, avoid]
                   | Exp "!=" Exp       [compare, avoid]

syntax priorites compare > 'Compare
```

This combination of productions combines to ensure that any comparison operator without paranetheses parses into a non-associative list, whereas the addition of parentheses causes multiple comparisons to be nested.

```
rule 'Compare(O:Object) => O

context Label:CompareKLabel(O:Object, HOLE)
when    notBool(isCompareKLabel(getKLabel HOLE))

context Label:CompareKLabel(O:Object,
        Label2:CompareKLabel(HOLE, _))

rule Label:CompareKLabel(O, Label2:CompareKLabel(O2, Rest))
  => Label(O, O2) and Label2(O2, Rest)
```

The two contexts above are what causes a multi-part comparison operator to only evaluate the middle terms once. The wrapper KLabel dissolves without modifying anything, but its presence in between two comparison operators prevents the contexts or the rule from applying, causing it to be evaluated independently of its sibling terms, as expected.

3.2.10 Assignment and Deletion

Assignment statements in Python come in several forms, based on the various different productions comprising assignment targets. The simplest and most basic of these is the assignment of an object to an identifier.

```
syntax Stmt ::= AssignTargets "=" Exp [strict(2)]
syntax AssignTargets ::= NeList{Target,"="}
rule (. => bind(X:Id, O)) ~>
     (X = Ts:AssignTargets => Ts) = O
```

As you can see here, assignment in Python evaluates first the object being assigned, then assigns it to each in a set of targets in order. The `bind` operation performs environment assignment, as covered in Section 3.2.7.

```
rule (. => bindList(Listify(Ts2), O, false)) ~>
     (Ts2:Targets = Ts:AssignTargets => Ts) = O:Object
rule (. => bindList(Listify(Ts2), O, false)) ~>
     ([ Ts2 ] = Ts:AssignTargets => Ts) = O
```

Assignment to a tuple or list of targets is performed using the `bindList` operator. We do not detail its semantics here, except to note the following:

- The object being assigned is treated as an iterable object and separated into a list of objects;
- The objects in the iterable are assigned to the targets in the tuple or list from left to right;
- A starred target accepts multiple objects and assigns to the target an object of type list;
- If there are too few or too many values in the iterable for the starred target to accept, a ValueError is raised.

```
context HOLE . _:Id = _ = K:Exp when isKResult(K)
rule (. => setattr(O, Id2String(X), O2) ;) ~>
     (O:Object . X:Id = Ts:AssignTargets => Ts) = O2:Object
rule invokeBuiltin(obj("setattr",_),
                   ListItem(O)
                   ListItem(Name:Object)
                   ListItem(Value:Object),
                   .)
  => getmember(O, "__setattr__", true, false, true)
         (Name, Value)
```

```
rule invokeBuiltin(obj("object.__setattr__",_),
                   ListItem(O)
                   ListItem(Name)
                   ListItem(Value),
                   .)
  => if isinstance(O, type):
       raise TypeError
     else:
       setmember(O, Name, Value)
rule invokeBuiltin(obj("type.__setattr__",_),
                   ListItem(Type)
                   ListItem(Name)
                   ListItem(Value),
                   .)
  => setmember(Type, Name, Value)
```

Assignment to an attribute reference calls the `setattr` function, after evaluating the object being dereferenced (but not the reference itself). The `setattr` function calls `__setattr__` on the object being assigned. `__setattr__` in turn uses the `setmember` operator. We do not detail the semantics of `setmember` except to say the following:

- We first check whether the attribute being assigned to is a data descriptor (i.e., has a `__set__` method;
- If present, the descriptor is invoked with the appropriate arguments;
- Otherwise, if a `__dict__` attribute is present, item assignment is invoked on the dict of the instance;
- If no `__dict__` attribute is present, an AttributeError is raised;
- If a `__dict__` is present but the object is a built-in type, a TypeError is raised instead.

```
context HOLE [ _ ] = _ = K:Exp
when isKResult(K)
context K [ HOLE ] = _ = K2:Exp
when isKResult(K) andBool isKResult(K2)
```

```
rule (. =>
     (getmember(O, "__setitem__", true, false, false) ->
          raise TypeError)
         (Key:Object, Value:Object) ;) ~>
     (O:Object [ Key:Object ] = Ts:AssignTargets => Ts) = Value
```

Item assignment executes the `__setitem__` function on the object being subscripted, after it and its subscript are evaluated. If the `__setitem__` attribute is not present, a TypeError is raised.

We will not discuss the semantics of object deletion (i.e., the `del` statement), except to note that it parallels the semantics of assignment quite closely. The primary difference is that `delattr` and `__delitem__` are used instead of `setattr` and `__setitem__`, and that starred targets are not allowed.

3.2.11 Class Statements

In essence, a class statement is an executable scope that populates the arguments to a call to `type.__new__`. The first step performed is to evaluate, sort, and unroll the positional, keyword, vararg, and kwarg arguments to a class declaration (see Section 3.2.8). A `metaclass` keyword parameter assigns an alternative metaclass to an object. The default metaclass is `type`. Positional arguments go to populate a tuple of base class objects. The remaining keyword arguments are stored in a map. At this point, if present, the `__prepare__` function is invoked on the metaclass to obtain a mapping to store local variables in. Then a new frame object is executed from the body of the class statement, after which the metaclass's constructor is invoked with the name of the class, the tuple of base classes, the populated locals dictionary, and any keyword arguments. Finally, the `__class__` free variable cell in the class scope is populated with the class that has been created.

However, this is only a partial accounting of the semantics of class statements, because we must also explain what invoking the constructor of the default metaclass, `type`, does. In essence, it performs a metaclass election by calculating the most derived metaclass which is a subclass of all the metaclasses of all the base classes provided. If this is not possible, a TypeError is raised. If the metaclass so calculated is a different metaclass than the `type`

```
rule LOCAL(_, Globals, Nonlocals, _, typeobject)
  => (SetItem("__class__") -Set Globals) -Set Nonlocals

rule FREE(Defs, Uses, Globals, Nonlocals, ChildFree, L)
  => Nonlocals ChildFree (((Uses SetItem("__class__"))
         &Set BOUND(.,L)) -Set Globals)
when    (Nonlocals &Set BOUND(., L)) ==Set Nonlocals
andBool "super" in Uses

rule BOUND(Bound:Set, L ListItem(scope(_, _:Set, _:Set, _:Set,
         Globals:Set, _:Set, _:List, typeobject, _)))
  => BOUND(Bound SetItem("__class__"), L)
when notBool "__class__" in Globals
```

Figure 3.7: Python Scoping Rules for Class Blocks

object, the constructor of the new metaclass is invoked instead. The constructor also checks whether Otherwise, the `__doc__`, `__module__`, `__bases__`, and `__dict__` attributes are assigned from the locals dictionary. Additionally, if `__eq__` is defined in the locals dictionary but not `__hash__`, we assign None to the `__hash__` attribute. We also decorate the `__new__` method with a `staticmethod` decorator if it is present. Finally, we call `type.mro` and assign the result to the `__mro__` attribute.

We will not discuss the semantics of the `mro` function except to explain that it follows the C3 method resolution order algorithm [23].

Note there are two abnormalities with respect to the way class scope operates which must be addressed in order to make our discussion of scoping (see Section 3.2.6) and environment lookup (see Section 3.2.7) complete.

To better understand these scoping rules, we provide rules pertaining to the scope of class blocks in Figure 3.7.

The first of these abnormalities occupies the first two of these rules, which pertain to the zero-argument `super()` call that was added in Python 3.0. Note in the rule that `__class__` is declared as both a FREE and LOCAL variable in class scopes (as denoted by the `typeobject` specifier in the rule). To better understand why this occurs, consider the following Python code:

```
class A:
  a = lambda self: super()
```

In this code, the body of the lambda is equivalent to the two-argument call

form `super(A, self)`. The first argument is constructed by means of adding a variable `__class__` as a member of `co_cellvars` to the class scope and as a member of `co_freevars` to the function scope. The second argument is constructed by means of obtaining the first argument to the function. In order for this to work, it is necessary that `__class__` be treated as a LOCAL variable in class blocks. It is also necessary that a use of the identifier `super` counts as a use of the identifier `__class__`.

One curiosity in particular to note is the side condition on the rule for BOUND. In the completed semantics, a `global __class__` declaration in a class scope would produce a SyntaxError. At this time, the generation of SyntaxErrors for scopes with invalid accesses and assignments is not part of the semantics, however, so we limit execution by providing only a partial evaluation for the BOUND function. The reason this is interesting is because a class scope with this particular global declaration causes a segmentation fault in the 3.3.2 final release of Python. Our detection of this behavior as a result of the testing of this semantics has led to a fix for this in the as-yet-unreleased Python 3.3.3.

The second abnormality of class scope can be seen in the rules in Figure 3.7 by means of examining the LOCAL and BOUND variables in a class scope. Aside from `__class__`, no other identifiers are BOUND or LOCAL in a class scope. To understand the meaning of this in practice, consider the following code block:

```
def a():
  class A:
    x = 5
    def b(self): return x
  x = 4
  A().b() # returns 4
```

Note here that the use of the variable `x` in a method inside a class refers not to the variable in the immediately enclosing class scope, but instead to the same variable in the nearest enclosed function scope, in this case the function `a`.

```
rule (. => moduleTarget(Module) =
          moduleToAssign(doImport(Module, .Aliases), Module)
     ) ~>
     import (Module:Alias , Rest => Rest)
rule import .Aliases => .

rule from Module:RelativeModule import Fromlist:Aliases
  => try:
       moduleTargets(Fromlist) =
           importFrom(doImport(Module, Fromlist),
               moduleNameIds(Fromlist))
     except AttributeError:
       raise ImportError

rule from Module import *
  => doImportStar(doImport(Module, String2Id("*"), .Aliases))

rule doImport(Module, Fromlist)
  => builtins . __import__ (
         moduleName(Module),
         globals(),
         locals(),
         None if Fromlist ==K .Aliases
             else (moduleNames(FromList),)
         moduleLevel(Module))
```

Figure 3.8: Import Statments

3.2.12 Import Statements

The semantics of an `import` statement in Python can be seen as divided into two major components. The first is the semantics of the statements themselves.

The rules in Figure 3.8 utilize a number of auxiliary functions which we do not explain in detail, but merely summarize instead their behavior. `moduleTarget` and `moduleTargets` obtain a single target (or list of targets) to assign the result of executing the `__import__` function to. This is obtained via either the name of the top-most module, or the alias it is referred to as. `moduleToAssign` obtains the value to assign to the target from the result of the `__import__` function. This is the same as the return value of the function, unless an alias is present, in which case it is the innermost module

in the dotted list of modules. `moduleNames` and `moduleNameIds` obtain the names of the attributes in the fromlist (as opposed to their aliased name). `importFrom` takes the list of names of attributes, and computes their dereference from the result of executing the `__import__` function. `moduleLevel` obtains the integral level of the relative import based on the number of dots preceding the module. `moduleName` obtains the string name of the module to be imported, by concatenating attribute references together. The module name is the empty string if the package is a relative package from the current directory with no name after the prefix of dots. Finally, `doImportStar` iterates through the `__all__`, if present, or the `__dict__`, if not, attribute of the value returned from the `__import__` statement, and assigns either all the attributes in the `__all__` list, or else all the attributes in the `__dict__` which do not begin with an underscore, to the local dictionary.

As you can see, then, each of the types of import statements ultimately calls the underlying implementation of `__import__` with the appropriate arguments, and processes the return value in order to assign bindings to the local environment. The `__import__` statements itself is merely that function with that name as present in the file `_bootstrap.py` in the `importlib` package in the Python standard library. Our semantics merely loads the AST for this module into the interpreter and executes it by passing the `_imp` and `sys` modules to its `_install` function.

3.2.13 Syntactic Sugar

A number of the remaining features of Python are implemented by means of transformations of one piece of syntax into another piece of syntax.

Assertions

```
rule assert E
  => if __debug__:
       if not E:
         raise AssertionError [macro]
rule assert E , E2
  => if __debug__:
       if not E:
         raise AssertionError(E2) [macro]
```

Slicing

```
rule : : => slice(None, None, None) [macro]
rule E : : => slice(E, None, None) [macro]
rule E : E2 : E3 => slice(E, E2, E3) [macro]
```

Slicing itself is implemented by means of the `slice` object and the `__getitem__`, `__setitem__`, and `__delitem__` methods of each sequence type. The rules shown above are only a few of the relevant ones. However, they are sufficient to show the pattern that the other rules also follow.

Decorators

```
rule declName(def X:Id ( _ ) -> _ : _) => X
rule declName(def X ( _ ) : _) => X
rule declName(class X ( _ ) : _) => X
rule declName(K newline _) => declName(K)


rule @ K
     K2:K
  => K2 newline declName(K2) = K(declName(K2)) [macro]
```

A decorator is merely equivalent to a function application following the thing decorated which takes the decorated object as parameter and assigns the result of the function to itself.

Comprehensions

```
rule [ E Comp ] => list(E Comp) [macro]
rule { E Comp } => set(E Comp) [macro]
rule { E : E2 Comp } => dict((E, E2) Comp) [macro]
```

List, set, and dict comprehensions are merely generator expressions oonverted using the appropriate constructor. Technically this is not how Python implements these constructs, because it instead more efficiently generates code that populates the objects directly. However, it is easy to see that this approach evaluates the same expressions in user code in the same order, and creates an equivalent result if no exception is raised. However, if an exception is raised by a comprehension, the traceback object for that exception

will differ depending on the approach used; therefore, we plan to alter the desugaring of these constructs in a future version of the semantics.

```
rule comprehension(E, for T in E2 Comps)
  => for T in E2:
       comprehension(E, Comps)
rule comprehension(E, if E2 Comps)
  => if E2:
       comprehension(E, Comps)
rule comprehension(E, .Comps) => yield E ;
```

A generator expression is merely an anonymous function that contains nested loops and conditionals inside of which is a yield expression.

3.2.14 OS Module

We do not implement the `os` module with any completeness. However, in order to support the basic features of `importlib`, it is necessary that we support a few of its functions. Because these functions are essentially wrappers placed around operating system calls, we do not detail the rules in the semantics itself for these functions. Instead, we describe the API the $\mathbb{K}$ framework exposes for these system calls.

```
syntax KList ::= "#stat" "(" String ")" [function]
               | "#lstat" "(" String ")" [function]
               | "#opendir" "(" String ")" [function]
syntax TCPError ::= "#EOF" | "#ENOENT" | "#ENOTDIR"
                  | "#EACCES" | "#ELOOP"
                  | "#ENAMETOOLONG" | "#EBADF"
```

Note here that the essential premise of the $\mathbb{K}$ framework design is that each of these system calls mirror a system call in POSIX. Thus, for example, `#stat` is based on `man 2 stat`, which takes a string path and returns a struct of values and a return value. In $\mathbb{K}$, the same is implemented by means of taking a string path and returning either a tuple of values, or an error code from `man 3 errno`. These system calls are implemented by means of a TCP socket being opened with a local server written in Java which performs native operations using Java 7's new I/O API, and returning error codes or return

values based on the exceptions thrown or values returned by the portable system call provided by the JVM. This mechanism provides a standard API for invoking system calls in $\mathbb{K}$ Framework tools. The server also manages sets of files with integer file descriptors in order to emulate the native file descriptors of a language like C without sacrificing managed code.

```
syntax Int ::= "#open" "(" String ")" [function]
syntax K ::= "#close" "(" Int ")" [function]
syntax Int ::= "#fReadByte" "(" Int ")" [function]
syntax String ::= "#fReadBytes" "(" Int "," Int ")" [function]
```

Note that `#open` takes a path and returns a file descriptor, `#close` closes a file descriptor, and `#fReadByte` and `#fReadBytes` take a file descriptor and read one or more bytes.

3.2.15 Dynamic Compilation

One final feature of note in the Python semantics is the ability to take a string and generate a code object from that string, then execute that code object. We do not go into the details of the `compile`, `eval`, and `exec` functions themselves, because they are not particularly interesting. Instead we focus only on the core functionality of `compile` itself which converts a string into a code object:

```
rule parseAndCompile(Source, _, "eval")
  => compile(return #parse(Source, "Exp"), "eval")
rule parseAndCompile(Source, _, "exec")
  => compile(#parse(Source, "Stmts"), "exec")
rule compile(#noparse,_)
  => raise SyntaxError
rule compile(return #noparse::K, _)
  => raise SyntaxError
```

Note that here Source is a String. The `#parse` operator is a built-in of the $\mathbb{K}$ framework which takes a String and a hint as to the sort of the string once parsed, and returns an AST of the parsed term of that sort, or `#noparse` if there is no valid parse of that sort for that string. Thus we compile a code object for an expression which returns the value of that expression, and a

```
#if S ==String "eval" #then
  eval(Exp, Globals, Locals, getEvalArgs(Exp))
#else
  eval(Exp, Globals, Locals, getEvalArgs(Exp)) ; ~>
  None
#fi

rule <k>
       eval(CO, Globals, Locals:Object, map(M:Map)) ~> K:K
       =>
       executeFrame(N, CO, ref(Frame), Locals, Globals,
           Globals["__builtins__"],
           makeCells(...),
           M) ~> return
     </k>
     <nextLoc> N => N +Int 1 </nextLoc>
     ...
```

Figure 3.9: `eval()` and `exec()` Core Implementation

code object for a set of statements which executes that set of statements. If parsing fails, we raise a SyntaxError.

The code snippet in Figure 3.9 is what executes the `eval` or `exec` functions, at their core. Note that if we are executing `eval` then we execute a frame and return the return value of that frame (None if it was compiled using "exec"). If we are executing `exec` then we execute a frame, discard the return value of the frame, and return None. `getEvalArgs` is a special function which computes the parameters to a function whose code object is being executed. If you execute the code object of a function directly, it does not populate default positional or keyword-only parameters, so any function with positional or keyword-only parameters throws a TypeError if its code object is executed. However, vararg and kwarg parameters are still successfully populated with the empty tuple and the empty dictionary if a code object with these is executed.

Chapter 4

Related Work

Research into the formal semantics of the Python programming language is a fairly new field. Aside from our work, the only extant attempts to formalize the semantics of Python that we could find are the `minpy` semantics found in [24] and the `lambda-py` semantics found in [13]. However, these are semantics of Python 2.5 and Python 3.2 respectively, whereas our own semantics implements Python version 3.3. As such, the three semantics are not strictly comparable. However, if we allow ourselves to modify the test suites so that they each pass all the tests on the appropriate version of CPython, we can use these modified test suites to create a bisimulation between our semantics and the two competitors.

We begin first with `minpy` because it is the older of the two.

4.1 Comparison with `minpy`

In order to compare our semantics with `minpy`, we first are required to construct a version of our test suite that passes CPython 2.5, and a version of `minpy`'s test suite that passes CPython 3.3. For the former, we took our test suite and ran it against CPython 2.5. Since a number of tests immediately began failing, we made several modifications, including:

- Class statements were given explicit base classes so as to avoid using old-style classes;
- Where the signature of an operation used by a test was changed between Python 2.5 and Python 3.3, the operation was modified to its original form;
- Syntax was changed where necessary to undo syntactic changes between Python 2.5 and Python 3.3;

- Where the exception raised by an operation (or the value that operation returns) were different than expected by the test, the expected value was changed;
- Where fields were added by Python 3.3 which did not exist in Python 2.5, assertions asserting the expected value of these fields were removed;
- Where tests tested features of Python 3.3 not present in Python 2.5, these tests were deleted;

Astonishingly, out of 236 tests that remained, all of which passed CPython 2.5, only ten tests passed `minpy`. We have not given a thorough accounting of all of these tests, however, we suspect that many, if not most, of the tests failed due to a failure to correctly identify the scope of a semantics of the Python language. `minpy` deliberately attempts to avoid implementing the semantics of built-in functions in the standard library whenever possible, and this is a mistake. Recalling that roughly 40% of the rules of our semantics are devoted to the implementation of built-in functions, and that the built-in functions are nowhere near complete despite most of the rest of the semantics being completed, it is clear that a semantics of Python that avoids implementing the standard library is a semantics that avoids implementing the vast majority of the language itself.

The second half of our evaluation of `minpy` consists of evaluating our own semantics against the test suite that they provide. In order to do this, it was necessary to convert the test suite from Python 2.5 to Python 3.3. This was achieved much more readily than the reverse conversion with the assistance of the 2to3-3.3 tool which performs automated conversion from Python 2.7 to Python 3.3. Once this tool had been run, only minor modifications were necessary in order to make the test suite pass: we accomplished this by means of a few modified assertions.

Ignoring the lack of tests for built-in functions, the `minpy` test suite is nonetheless quite extensive. It handles a number of quite specific corner cases in the language specification that might not otherwise come to light. However, when ran against our interpreter, the results are nonetheless quite impressive. Of 139 tests, 130 tests pass our specification. In fact, when considering the 9 tests that do not pass, 7 of these correspond to tests we excluded from our test suite because they would take too long to execute,

and the remaining two are tests of the `print` function, which we have not yet implemented. Thus, given sufficient time and only a marginally more complete semantics, we fully expect that we would be able to pass the entire test suite associated with the `minpy` semantics. This demonstrates quite clearly that our semantics very nearly implements a superset of the semantics of `minpy`.

4.2 Comparison with `lambda-py`

In order to compare our semantics with `lambda-py`, we had the considerably easier task of converting the two test suites (`lambda-py`'s and our own) between version 3.2 and 3.3. Again, this was accomplished by means of modifications of assertions and other similar changes.

When tested against the 266 tests in our test suite (note the larger number than the test suite for `minpy` is because we had to delete some tests earlier because they were not relevant to Python 2.5), we find again that a very substantial number of our own tests fail in the `lambda-py` semantics. To be precise, 107 tests passed, 155 tests failed, and 4 tests timed out (we do not presume to modify the test harness for `lambda-py` to attempt to debug this beahvior). Examining those tests that failed, we find that 30 tests failed because a callable type did not possess a `__call__` method, 19 tests failed because a built-in function was not implemented, 20 tests failed because a built-in object did not have a necessary method, 12 tests failed because an AST node was not implemented, and 14 tests failed because a built-in module was simply not implemented at all.

Most tellingly, however, some 60 tests failed with incorrect behavior or failed assertions. In these cases, the semantics was not merely incomplete, but in fact entirely mistaken. Calls succeed which should have failed, calls fail which should have succeeded, and others still simply return entirely incorrect results. The blame for this once again is solely at the feet of the design of the semantics: it implements much of the standard library, where it does at all, using pure Python code, despite the fact that many features of many built-in functions are simply not expressible in this fashion, because they rely on underlying primitives that Python does not expose.

Considering the reverse case, an analysis of the behavior of our own se-

mantics against the test suite of `lambda-py`, we find ourselves behaving more favorably. Of the 224 tests in their test suite, our own semantics passes 186 of them, failing only 34 tests (plus an additional 4 which were excluded due to time constraints, because they would require a considerable time to execute). Furthermore, when analyzing these failures, we find that they are entirely the result of the failure to implement 2 built-in types, 5 built-in functions, and 22 built-in methods on types we already partially define.

As is evident, we are by no means complete in our implementation of a semantics of Python 3.2. However, it is clear from the evidence that our attempt provides a more complete semantics of the language than any that comes previous. Where it is incomplete, it is incomplete as a result of a lack of sufficient time to implement the relevant built-in functions, rather than due to any simplification or incorrectness in the rules that we provide. And we feel confident that in time we will be able to overcome even these deficiencies in its quality.

Chapter 5

Conclusion

In the previous chapters, we demonstrate that a formal semantics for Python 3.3 can be constructed in the $\mathbb{K}$ framework, and that, when complete, it will serve as the first *complete* semantics of its kind, rather than simplifying itself in any way. We now discuss some of the limitations in the scope of the work, such as the pieces that remain incomplete, and the future work that is needed to support verification using this semantics.

5.1 Limitations and Future Work

Here we discuss several of the features of Python which were neglected in this semantics for reason of time constraints, as well as providing detail of several ways in which the undefinedness of the semantics can be improved.

The obvious way in which the semantics is incomplete can be seen in a lack of features. We have failed to provide a semantics to the new feature of `yield from`. We have failed to implement private name mangling of variables in class definitions. We have not implemented any concern for futures statements (although arguably such support is unnecessary until such a time as Python 3.4 is released). We have not fully implemented a very significant number of built-in functions, built-in types, and built-in modules. Indeed, we have made very little effort to capture the syntax of Python either, relying on CPython's own parser to do much of our work for us. In this and several other ways, much of the functionality of Python 3.3 is lacking from our semantics.

However, there is another way in which our semantics could be improved. This is by increasing the degree to which it is capable of expressing that a particular program in Python is not well-defined. While Python does not have any rigorous notion of well-definedness in the sense of there being operations which are considered unsafe and can cause arbitrary undesirable side

effects, nonetheless there are a number of cases where it seems clear that it would be desirable to treat the end result of a particular computation as *not specified*; that is to say, as not being guaranteed to produce any particular value. For example, while the Python language specification guarantees that `co_consts[0]` of a code object is the object's docstring, it provides no guarantee as to the contents or even the length of the rest of the tuple. Similarly, the Python language does not provide any guarantee that garbage collection will occur at any particular time or even at all. However, it does guarantee that when garbage collection occurs, if a `__del__` method exists on an object then it will be invoked. Or again, Python provides no guarantee as to whether `5 is 5` returns True or False. There are many cases of this in our semantics. We would like, in the long run, to be able to perfectly capture the semantics of these operations by means of the construction of boolean postconditions on symbolic constants. However, for the time being, all of these operations are overspecified. For example, we do not ever invoke a `__del__` method, `co_consts` is always of length 1, and `5 is 5` returns True.

All of these features represent opportunities for future work on the semantics of Python, which we hope to employ in the long run in order to generate a fully complete and appropriately specified semantics of Python which is usable for formal reasoning. However, as it presently stands, our semantics is already a formidible improvement over the status quo in research into the formal semantics of the Python family of languages.

References

[1] F. Chen and G. Roşu. Rewriting Logic semantics of Java 1.4, 2004. URL `http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java`. 2

[2] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350. Springer, 2007. 7

[3] Corporation for National Research Initiatives. The Jython project, Feb. 2013. URL `http://www.jython.org/`. 6

[4] T. F. Şerbanuţă and G. Roşu. K-Maude: A rewriting based tool for semantics of programming languages. In P. C. Ölveczky, editor, *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010*, volume 6381 of *Lecture Notes in Computer Science*, pages 104–122, 2010. ISBN 978-3-642-16309-8. doi: doi:10.1007/978-3-642-16310-4_8. 2, 7

[5] T. F. Şerbănuţă and G. Roşu. KRAM—extended report. Technical Report `http://hdl.handle.net/2142/17337`, UIUC, September 2010. 2

[6] C. Ellison. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois, July 2012. 2

[7] A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of java programs in JavaFAN. In *Proceedings of Computer-aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 501 – 505, 2004. 2

[8] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF—reference manual—. *SIGPLAN Not.*, 24(11): 43–75, 1989. doi: 10.1145/71605.71607. 7

[9] M. Hills and G. Roşu. KOOL: An application of rewriting logic to language prototyping and analysis. In *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA'07)*, volume 4533 of *LNCS*, pages 246–256. Springer, 2007. 2

[10] IronLanguages. Ironpython.net, July 2012. URL `http://ironpython.net/`. 6

[11] P. O. Meredith, M. Katelman, J. Meseguer, and G. Roşu. A formal executable semantics of Verilog. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)*, pages 179–188. IEEE, 2010. doi: doi:10.1109/MEMCOD.2010.555863. 2

[12] G. R. Patrick Meredith, Mark Hills. An executable Rewriting Logic semantics of K-Scheme. In D. Dube, editor, *Proceedings of the 2007 Workshop on Scheme and Functional Programming (SCHEME'07), Technical Report DIUL-RT-0701*, pages 91–103. Laval University, 2007. 2

[13] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full Monty. In *Proceedings of the 28th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'13)*. ACM, October 2013. 41

[14] PyPy. Pypy, June 2013. URL `http://pypy.org/`. 6

[15] Python Software Foundation. History and license, July 2013. URL `http://docs.python.org/3/license.html`. 6

[16] Python Software Foundation. Special method lookup, July 2013. URL `http://docs.python.org/3/reference/datamodel.html#special-method-lookup`. 19

[17] G. Roşu. K tutorial, Oct. 2012. URL `http://k-framework.org/index.php/K_Tutorial`. Accessed July 3, 2013. 11

[18] G. Roşu, A. Ştefănescu, c. Ciobâcă, and B. M. Moore. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*. IEEE, June 2013. 8

[19] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi: 10.1016/j.jlap.2010.03.012. 2, 7, 11

[20] G. Rosu, W. Schulte, and T. F. Serbanuta. Runtime verification of C memory safety. In S. Bensalem and D. A. Peled, editors, *Runtime Verification (RV'09)*, volume 5779 of *Lecture Notes in Computer Science*, pages 132–152, 2009. doi: doi:10.1007/978-3-642-04694-0_10. 2

[21] T. F. Şerbănuţă. *A Rewriting Approach to Concurrent Programming Language Design and Semantics*. PhD thesis, University of Illinois at Urbana-Champaign, December 2010. `https://www.ideals.illinois.edu/handle/2142/18252`. 2

[22] T. F. Serbanuta, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Rosu. The K primer (version 2.5). In M. Hills, editor, *K'11*, Electronic Notes in Theoretical Computer Science, to appear. 2, 7

[23] M. Simionato. The Python 2.3 method resolution order. URL `http://www.python.org/download/releases/2.3/mro/`. 33

[24] G. J. Smeding. An executable operational semantics for Python. Master's thesis, Universiteit Utrecht, January 2009. `http://gideon.smdng.nl/wp-content/uploads/thesis.pdf`. 41

[25] TIOBE Software BV. TIOBE programming community index, June 2013. URL `http://www.tiobe.com/index.php/content/paperinfo/tpci/`. 6

[26] D. N. Welton. Programming language popularity, Apr. 2011. URL `http://langpop.com/`. Accessed July 3, 2013. 6