Thesis for the degree of Master of Science

An executable operational semantics for Python

Gideon Joachim Smeding

January, 2009
INF/SCR-08-29

Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Universiteit Utrecht
Utrecht, The Netherlands

Supervisors:
dr. Andres Löh
prof. dr. S.D. Swierstra

Abstract

Programming languages are often specified only in an informal manner; in the available documentation, the language behavior is described by examples and text. Only the implementation, a compiler or interpreter, describes the exact semantics of constructs.

Python is no different. It is described by an informal manual and a number of implementations. No systematic, formal descriptions of its semantics are available.

We developed a formal semantics for a comprehensive subset of Python called *minpy*. The semantics are described in literate Haskell. The source files are compiled to an interpreter as well as the formal specifications in this document. In a sense, this document is the documented source code of the interpreter.

Contents

1 Introduction **7**
1.1 Python . . . 7
1.2 Formal semantics . . . 8
1.3 Contributions . . . 10
1.4 Overview . . . 10

2 Preliminaries **11**
2.1 Lists . . . 11
2.2 Mappings . . . 11

3 Abstract syntax **13**
3.1 Expressions . . . 13
3.2 Statements . . . 14

4 Transforming an abstract machine **15**

5 Theory of objects and classes **17**
5.1 The method resolution order . . . 17
5.2 Mappings of inherited attributes . . . 20
5.3 Classes as types . . . 21

6 Basic rewrites rules **23**
6.1 Blocks . . . 23
6.2 Pass and expression statements . . . 24

7 Variables **25**
7.1 Assignment statements . . . 25
7.2 Variable expressions . . . 26
7.3 Delete statements . . . 26

8 Functions and generators **27**
8.1 Defining functions and generators . . . 27
8.2 Executing functions . . . 29
8.3 Executing generators . . . 33

9 Classes and objects **37**
9.1 Class definition . . . 37
9.2 Object creation . . . 39
9.3 Attribute access . . . 41
9.4 Attribute assignment . . . 42
9.5 Attribute deletion . . . 43

10 Exceptions **45**
10.1 The raise statement . . . 45
10.2 The try-except statement . . . 47
10.3 The try-finally statement . . . 48

11 Control structures **51**
11.1 The if statement . . . 51
11.2 The while statement . . . 52
11.3 The for statement . . . 54
11.4 The break statement . . . 55
11.5 The continue statement . . . 56

12 Print statements **57**

13 Operators **59**
13.1 Unary operators . . . 59
13.2 Binary operators . . . 60
13.3 Container operators . . . 62

14 Dynamic code execution **65**
14.1 The exec statement . . . 65
14.2 The import statement . . . 66

15 Conclusion **69**
15.1 Literate programming . . . 69
15.2 The interpreter . . . 70
15.3 The test suite . . . 72
15.4 Future work . . . 73

A Builtin values **75**
A.1 The base objects `object` and `type` . . . 75
A.2 The function and generator . . . 76
A.3 Integer values . . . 76
A.4 Booleans . . . 78
A.5 Strings . . . 78
A.6 Lists . . . 80
A.7 Tuples . . . 82
A.8 The tuple class . . . 82
A.9 Dictionaries . . . 83

1 Introduction

This introduction is organized as follows. First we review Python with respect to formal semantics, and define a scope for our work. Then we discuss executable operational semantics and our approach to it. Finally, we list the main contributions, and give an overview of the rest of this thesis.

1.1 Python

Python is an imperative, dynamic, object-oriented programming language originally developed by Guido van Rossum at the CWI in the Netherlands in the 1980s. Since the first publication of the code in 1991, Python has gained much popularity and now is a major programming platform.

A formal semantics for Python

There are no formal semantics for Python. CPython is the de facto reference implementation. Although it maintains high coding standards, CPython is not written with legibility as its primary focus.

The PyPy [11] implementation of Python is written in Python itself. To be exact, it is written in a restricted form of Python called RPython [1]. The introduced restrictions facilitate static analyses such as typing, and enable good run-time efficiency.

Because it is easy to read for Python programmers, it has been suggested that PyPy might one day become the reference implementation of Python. However, even for the restricted form of Python, no systematic documentation of the operational semantics exists.

Despite the lack of a formal semantics, Python appears to be a particularly good candidate for formal specification, for a number of reasons.

- Python is a relatively simple language. It has a limited number of constructs, most of which are found in many other languages. Python introduces no radically new features, but has a unique combination of features found in its predecessors.
- Despite being a simple language, it is a widely used language which is still gaining in popularity. As a language and platform, Python has proven to be mature: many sizable projects use it to great success.
- The language is quite well documented. There is an extensive reference manual [15] and all past changes to the language have been documented in Python enhancement proposals (PEPs) [14].
- There are a number of independent, mature implementations of Python available [3, 6, 7, 11]. Semantics can be used to compare interpreters and maintain language conformism.

These semantics target Python version 2.5. The behaviour of the CPython implementation, not the documentation, is taken as the definition of Python.

Scope of the semantics

Of course we would have liked to document all of Python, but time constraints did not allow that. Therefore the scope of these semantics is limited to a subset of Python that we call *minpy*. The language *minpy* includes virtually all syntactical constructs of Python, but leaves out much of the syntactic sugar (see Chapter 3 for an overview of the abstract syntax).

While most of the concepts introduced by the syntax are included in these semantics, much of the language's features have been ignored:

- The standard library has not been specified wherever possible. Some elements of the library are needed to support other features. For example, exception handling naturally requires the Exception class.

- Garbage collection is not modeled by these semantics. Apart from managing the storage of objects in memory, garbage collection affects the operational semantics by calling an object's finalizer, which can be specified by a programmer. Typically different implementations have their own garbage collectors and some even allow a user to control it.

- Python's support for multi-threading is ignored. While multi-threading is an interesting subject by itself, it is simply beyond the scope of this project. It could also be argued that threads are not truly part of the language, since they are only supported through the interpreter's libraries.

- The ability of Python to interact with the 'outside world' is not described. In practice, the 'outside world' is represented by other libraries. Interaction between two different languages is achieved through what is sometimes called a foreign function interface.

- The specification of simple types in this document, such as integers and strings, do not describe the exact behaviour of CPython (see Chapter A). The semantics of simple types tend to differ in the small details. Types and their associated operators usually have the semantics of the platform underpinning the interpreter. For example, Jython uses Java's integers while CPython's integers are based on C's integer types.

- These semantics only work with the so called new style classes that have been introduced in CPython version 2.2 [4]. The old-style classes had been kept for backwards compatibility only, and will not be in future versions of Python.

- Python has various reflexive features. For example, it is possible to inspect the dictionary that implements objects, and the body of a function can be replaced at runtime. These features are not covered by our semantics.

1.2 Formal semantics

Formal semantics seek to precisely describe the meaning of a programming language using mathematical constructs. All aspects of programming languages can be formally specified, but usually a formal semantics describes the run-time behaviour of programs.

Operational semantics

A number of different styles of formal semantics exist. Operational semantics as introduced by Plotkin [10], describe language semantics in terms of a state transition system. Using an abstract machine for the state, the operational semantics will closely resemble an interpreter. Because a state machine uses well known constructs like stacks and heaps, the semantics will be comparatively easy to understand.

In general, formal semantics improve our understanding of a language to its finest details, simplifying reasoning about programs at an abstract level. Operational semantics are of significant practical value as well:

- The detailed documentation provided by an operational semantics facilitate the creation of interpreters of compilers. Especially if accompanied by a test suite, standardization is much simpler to achieve and maintain.
- Program analyses can be designed in a more systematic fashion and formal reasoning can be used to verify properties of analyses. More generally, many language tools are easier to develop with an operational semantics at hand.
- A language can be extended and altered more safely, since interaction of extensions with the original language can be analyzed systematically. For example backwards compatibility can be guarded more closely.

While the case for operational semantics is easy to make, some disadvantages of formal semantics in general also apply to operational semantics, albeit to a lesser degree.

- Formal semantics are often perceived to be an ivory tower matter, i.e., be of little value to neither the users nor the developers of a language. The following anecdote illustrates this nicely. When inquiring after previous work on formal semantics on the Python mailing list, someone answered "[...] I don't think Python culture operates like that very much."
- Formal semantics are often defined separately from the actual implementation. Whether or not a compiler or interpreter actually adheres to the defined semantics is unclear and difficult to judge. Very few compilers or interpreters have been proven correct with respect to the semantics.
- It has been argued that formal semantics might inhibit evolution of a language [5]; the maintenance costs of formal semantics could discourage experimentation and extension.

An executable operational semantics

The listed shortcomings of formal semantics can be eliminated by making the operational semantics executable. In other words, by writing the semantics in such a way, that it can be compiled into an interpreter.

An executable operational semantics would be of immediate practical value as an interpreter. Furthermore it simplifies maintenance of the operational semantics, as the semantics and implementation are developed simultaneously. An executable operational semantics enables experimentation and simplifies language evolution.

The operational semantics in this document have been written in the programming language Haskell, or more specifically literate Haskell, a variant that allows Haskell code to be mixed with LaTeX. The Haskell code has been formatted using lhs2TeX[9] and some simple scripts. Thus the sources used to produce this document, can also be compiled into an interpreter for *minpy*.

The source code has been reformatted in such a way, that no knowledge of Haskell is required. One might even forget that the semantics were written in a programming language at all, as the rules look like common mathematical equations.

Normally writing the semantics of a complete language is a tedious and error prone process. Writing an executable operational semantics on the other hand is exciting, as one's progress is clearly reflected in the interpreter. The strong type system of Haskell prevents many kinds of mistakes, and we can compare the behaviour of our executable semantics to that of Python with test cases. While testing cannot prove the correctness of the semantics, it does justify confidence in the semantics' quality.

1.3 Contributions

The main contributions of this master's thesis are the following:

1. Firstly, we have described the operational semantics of a significant subset of Python;
2. secondly, we have created an interpreter from the same sources as the operational semantics;
3. and finally, we have created a test suite that compares the behaviour as described by our operational semantics to the CPython implementation, or other implementations.

1.4 Overview

This thesis is organised as follows. First, we set the stage for the semantics by introducing some notational conventions in Chapter 2, describing the abstract syntax of *minpy*in Chapter 3, and introducing the abstract machine model in Chapter 4. Then, in Chapter 5 we introduce the object model of *minpy*.

The remaining chapters, except for the conclusion, contain the semantic rules that describe the behaviour of specific statements and expressions. We start with some basic rules in Chapter 6, followed by the rules describing variables in Chapter 7, functions and generators in Chapter 8, classes and objects in Chapter 9, exception handling in Chapter 10, control structures in Chapter 11, printing in Chapter 12, operators in Chapter 13, and finally the exec and import statements in Chapter 14.

Finally, in Chapter 15, we discuss the results of this work and the lessons that we learned in the process.

2 Preliminaries

Before discussing the semantics themselves we introduce the notational conventions used for lists and mappings.

2.1 Lists

In these semantics we use many lists. Lists are sequences which are ordered collections of elements. For example, a list $[3, 5, 2, 5]$ contains the elements 3, 5, 2, and 5 in that order. Lists can also be empty, which would be denoted as pair of empty brackets.

By convention we overline names of lists, e.g. $\overline{a}$ would be a list of addresses. Individual elements of $\overline{a}$ are referred to by their index starting at 1. For example, q_2 refers to the number 4 in the list $\overline{q} = [1, 4, 3]$. The length of a list $\mid \overline{q} \mid$ is defined as the number of elements in the list. The operator : prepends a list with a new element and the operator $+\!\!+$ concatenates two lists.

2.2 Mappings

A mapping is a partial function that maps keys to values. For example, arrays and hash tables are mappings. Python itself has native support for mappings, which it calls dictionaries.

We use a number of special operators and notations to describe, alter and inspect maps. These operators have the same semantics for all mappings, regardless of their specific contents.

The list of key-value pairs $[k_1 \mapsto v_1, ..., k_n \mapsto v_n]$ represents a mapping m of keys $k_{1...n}$ to values $v_{1...n}$ respectively. Each key $k \in m$ maps to a value $v = m(k)$. Thus, the key-value pairs in the mapping are unique. An empty mapping is denoted as $\emptyset$, which is never used for empty sets.

The operator $\oplus$ combines two mappings A and B to create a new mapping that consists of the key-value pairs of both A and B. For duplicate keys, the key-value pairs of the left hand side of $\oplus$ take precedence over the right hand side. Formally defining this operator, we have:

$$(A \oplus B)(k) = \begin{cases} B(k) & \text{if } k \in B \\ A(k) & \text{otherwise} \end{cases}$$

The operator $\ominus$ removes a key-value pair, identified by its key, from a mapping. If the key is not in the domain of the mapping, nothing changes. Formally defining this operator, we have:

$$(A \ominus k)(j) = \begin{cases} A(j) & \text{if } k \neq j \\ \bot & \text{otherwise} \end{cases}$$

3 Abstract syntax

This chapter describes the abstract syntax of *minpy*. First we will introduce the syntax of expressions and operators, followed by statements and blocks. The semantics of each syntactical construct will be described in the following chapters.

The syntax definitions in this chapter are reformatted Haskell data definitions that mimic concrete Python syntax. Lists of a syntax element A are denoted as $\langle A \rangle^*$.

Because the concrete syntax of Python has already been specified by the reference manual [13], it will not be discussed here. The parser used by the *minpy* interpreter only implements a minimal subset of Python's concrete syntax.

The abstract syntax clearly shows some of the limitations of *minpy* compared to full Python. For example, list comprehension expressions are missing and try-except statements are limited to a single except clause. To keep the specification simple, we have made various optional elements mandatory. For example, the else branch in the if-then-else statement is not optional in this abstract syntax.

3.1 Expressions

The expression syntax specification indicates no binding preferences. Any ambiguity will be resolved by parenthesis. For example $3 + 2 * 5$ will have to be written as $3 + (2 * 5)$.

```
Expr ::= Name                  -- Variable , see Rule 7.3
       | Expr(⟨Expr⟩*)         -- Function call , see Rule 8.2
       | Expr.Name             -- Attribute access , see Rule 9.10
       | Expr[Expr]            -- Slice access , see Rule 13.11
       | Expr BinOp Expr       -- Binary operator , see Rule 13.5
       | UnaryOp Expr          -- Unary operator , see Rule 13.1
       | yield Expr            -- Yield expression , see Rule 8.13
       | Int                   -- Literal integer , see Rule A.5
       | Bool                  -- Literal boolean , see Rule A.10
       | String                -- Literal string , see Rule A.13
       | [⟨Expr⟩*]             -- Literal list , see Rule A.21
       | (⟨Expr⟩*)             -- Literal tuple , see Rule A.30
       | {⟨Expr : Expr⟩*}      -- Literal dictionary, see Rule A.35
```

```
BinOp ::= + | - | * | / | % | ** | // |==|! =|<|<=|>|>=| is | in | and | or
```

```
UnaryOp ::= not | -
```

3.2 Statements

The command line interface of Python accepts single statements. Full programs as well as modules, consist of a single block of statements –a list of statements– in a text file.

We often append a semicolon to statements to distinguish statements, and especially expression statements, from expressions. The concrete syntax of Python supports this use of semicolons as well, but does not require it.

Some statements, so called compound statements, contain blocks. In the concrete syntax these blocks are delimited by their indentation level (see Python's reference documentation). In this abstract syntax we delimit blocks using curly brackets where ambiguity arises. Specifically the empty block is denoted as an empty pair of curly brackets.

Stmt ::=	*Expr*	-- expression statement, see Rule 6.5
	Name = *Expr*	-- variable assignment , see Rule 7.1
	Expr.*Name* = *Expr*	-- attribute assignment, see Rule 9.15
	Expr[*Expr*] = *Expr*	-- slice assignment , see Rule 13.14
	`del` *Name*	-- variable deletion , see Rule 7.4
	`del` *Expr*[*Expr*]	-- slice deletion , see Rule 13.18
	`del` *Expr*.*Name*	-- attribute deletion , see Rule 9.19
	`def` *Name*(⟨*Name*⟩*) : *Block*	-- function definition , see Rule 8.1
	`return` *Expr*	-- return , see Rule 8.8
	`print`(*Expr*)	-- print , see Rule 12.1
	`if` *Expr* : *Block* `else` : *Block*	-- if-then-else , see Rule 11.1
	`while` *Expr* : *Block*	-- while , see Rule 11.4
	`for` *Name* `in` *Expr* : *Block*	-- for , see Rule 11.8
	`class` *Name*(*Expr*) : *Block*	-- class definition , see Rule 9.1
	`try` : *Block* `except` *Expr*`,` *Name* : *Block*	-- try-catch , see Rule 10.6
	`try` : *Block* `finally` : *Block*	-- try-finally , see Rule 10.10
	`raise` *Expr*	-- raise , see Rule 10.1
	`import` ⟨*Name*⟩*	-- import , see Rule 14.6
	`pass`	-- pass , see Rule 6.4
	`break`	-- break , see Rule 11.14
	`continue`	-- continue , see Rule 11.15
	`exec` *Expr* `in` *Expr*	-- exec , see Rule 14.1

Blocks in functions, classes and modules also define a variable scope, the semantics of which are discussed in chapters 7, 8, 9, and 14. There are some restrictions on the syntax of scoping blocks that are not expressed by the abstract syntax definitions. It could be argued that these are not syntactical restrictions, but Python raises a `SyntaxError` for programs that fail the restrictions:

- The return and yield statements are limited to the scope of a function. In other words, they can only occur in function definitions or in a while, for, try, or if statement inside the function.
- A function scope can contain either a return or a yield statement, but not both.
- The continue and break statements can only occur in for or while loops, or in a nested if or try statement.

4 Transforming an abstract machine

The operational semantics are defined by state transitions of an abstract machine. A machine state $\langle \Theta, \Gamma, S, \iota \rangle$ consist of a heap Θ, an environment stack Γ, a control stack S, and an instruction ι. State transitions are defined by rewrite rules that transform the machine state. These rewrite rules have the following shape.

$$\langle \Theta, \Gamma, S, \iota \rangle \Rightarrow \langle \Theta', \Gamma', S', \iota' \rangle$$

As the semantics of Python are deterministic, there is one, and only one, rule for each machine state. The states are discerned by pattern matches in the rules. In some cases however, pattern matches overlap. In these cases, the most specific pattern takes precedence over the others.

Heap

The heap is a mapping of addresses to values. Addresses are represented by natural numbers. Values include integers, strings, functions and objects.

For example, a heap $\Theta = [a_1 \rightarrow \texttt{"spam"}, a_2 \rightarrow 3, a_3 \rightarrow [a_1, a_2]]$, contains a string, a number, and a list at the addresses a_1, a_2, and a_3 respectively. A new heap $\Theta' = \Theta \oplus [a_1 \rightarrow \texttt{"eggs"}, a_4 \rightarrow 42]$ is defined as an update of the heap Θ: the string at a_1 is redefined and the value 42 is added at address a_4.

Values are never removed from the stack. Because of this, the operational semantics presented here will never define a practical interpreter. In a full implementation of Python, unused values will be removed from the heap by a garbage collector.

Environment

The environment is a stack of addresses. The addresses point to environment mappings on the heap, each of which contains the bound variables in a single scoping block. Environment mappings are values that implement a mapping of variable names -represented by strings- to addresses.

The environment mappings on an environment stack can be merged to create the environment mapping Σ^{Θ}_{Γ}, that contains all variable bindings of the environment mappings on the environment stack. The merged environment mapping Σ^{Θ}_{Γ} is defined as:

$$\begin{aligned} \Sigma^{\Theta}_{\square} &= \emptyset \\ \Sigma^{\Theta}_{\Gamma|\gamma} &= \Sigma^{\Theta}_{\Gamma} \oplus \Theta(\gamma) \end{aligned}$$

For example, the environment Γ with a heap Θ binds the variables x and y to the values 1 and 2 respectively. The variable x is bound in both γ_1 and γ_2, but the binding in γ_2 shadows the binding in γ_1, i.e., $\Sigma^{\Theta}_{\Gamma}(x) = a_2$.

$$
\begin{aligned}
\Gamma &= \square|\gamma_1|\gamma_2 \\
\Theta &= [\, a_1 \to 1 \\
&\quad\ , a_2 \to 2 \\
&\quad\ , \gamma_1 \to [\texttt{"x"} \to a_2, \texttt{"y"} \to a_2] \\
&\quad\ , \gamma_2 \to [\texttt{"x"} \to a_1] \\
&\quad\ ]
\end{aligned}
$$

Control stack

The control stack is a stack of continuation frames, i.e., frames that indicate 'what to do next'. Many frames use a placeholder $\circ$, to indicate what part of an instruction is currently being executed or evaluated.

For example, the stack $\square|x = \circ$ consists of a single frame $x = \circ$ on top of the empty stack $\square$. The circle in the frame $x = \circ$ replaces the assignment's right hand side to indicate that it is being evaluated. Once the right hand side expression has been evaluated, the assignment is executed.

Instruction

The instruction is a kind of program counter for the virtual machine. There are three different kinds of instruction: The instruction can be the expression, statement, or block that is being executed; it can be the result of the previously executed instruction; or an unwind instruction.

Side effects

Some rules have an effect on a hidden state not part of the abstract machine. Specifically I/O operations, such as printing to the screen, are typical side effects.

Because the outside world is not modelled in these semantics, side effects are informally described over the arrow of the rewrite rule. For example, a side effect q of a rule would be denoted as follows.

$$\langle \Theta, \Gamma, S, \iota \rangle \overset{q}{\Rightarrow} \langle \Theta', \Gamma', S', \iota' \rangle$$

The side effect q occurs when the machine state is rewritten. Rules are never selected based on a side effect, i.e., the pattern match of a rewrite rule can not be influenced by the side effect of the rewrite rule.

The import statement (see Section 14.2) has no side effects, even though reading files usually has side effects. We assume however, that the imported files do not change during the execution of a program.

Source of a rewrite rule

The rewrite rules are written in Haskell and reformatted to yield formulas of the above form. For example, the source of Rule 6.3 is as follows:

```
rewrite          (State  heap  envs  ( stack :|: BlockFrame b )  ( BwResult  a )) = case b of
  EmptyBlock  ->  (state  heap  envs  ( stack                 )  ( BwResult  a ))
  otherwise   ->  (state  heap  envs  ( stack                 )  ( FwBlock   b ))
```

5 Theory of objects and classes

Python is an object-oriented language. Values, including built-in values such as integers, strings, and functions, are represented by objects. Objects are mappings of names to addresses. The addresses point to the object's data and the associated operations (functions).

Every object has exactly one class. Even classes, being objects themselves have a class. The class-of relations between object and their classes can be represented by a tree. At the root of such a tree is the class `type`, which is its own class. See for example Figure 5.1.

Every class has one or more base classes, with the exception of the object `object`, which has no bases itself. The bases of a class are sorted by their local precedence order. The bases of a class A, the bases of the bases, and so on, are the superclasses of A. The object `object` is a superclass of all classes.

The base-of relation between types can be represented by a directed (from class to base), acyclic graph with edges ordered by their local precedence order. In this inheritance graph, all paths eventually lead to the base object `object`. See for example Figure 5.2.

Both the class and bases are object attributes: the class of an object is stored in the attribute `__class__`, and the `__bases__` attribute contains a tuple of a class' base classes. The order of bases in the `__bases__` tuple define the local precedence order. Classes are differentiated from other objects, only by their `__bases__` attribute, which 'normal' objects lack.

All values $\Theta(a)$ on a heap Θ have an object representation Ω_a^Θ. The object representation is the mapping of names to values, i.e. the object's attributes. The value of an object is determined by the class of an object. For example, objects of class `type` or `object` have a value `None`, and objects of class `int` have a primitive integer value.

The object representations of built-in types and functions, are not stored on the heap. Built-ins have an immutable object representation $[\,\texttt{\_\_class\_\_} \rightarrow a_C\,]$, where C is the class of the built-in type. Only a module value returns an object representation extended with its own mapping.

5.1 The method resolution order

A class inherits the attributes of its superclasses. Objects have access to the attributes of their class, including the inherited attributes, as if they are their own attributes. However, when classes have different definitions for the same attributes, it is unclear which takes precedence.

To resolve conflicts between inherited attributes, we define a so called method resolution order (MRO). The order of classes in the MRO determines which attribute overrides the other attributes of the same name. Despite the name, the MRO determines not only the resolution of methods (function attributes), but the resolution of all attributes.

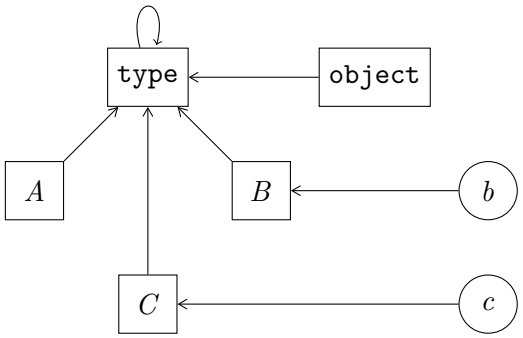

Figure 5.1: Example tree showing class-of relations. The classes `object`, A, B, and C have class `type`, which is its own class. The objects a and b have classes A and B respectively.

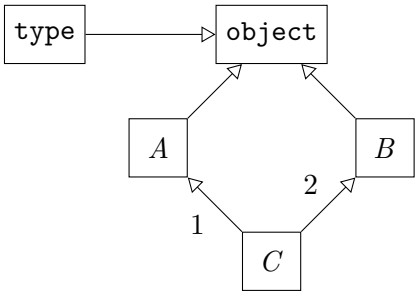

Figure 5.2: Example inheritance graph. The classes `type`, A, and B inherit from `object`. The class C inherits A and B, in that order.

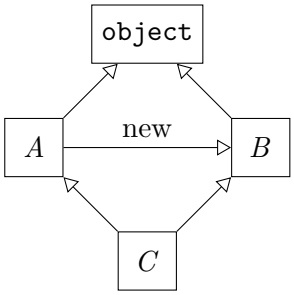

Figure 5.3: A (partial) extended inheritance graph. The marked arrow is added to the inheritance graph in Figure 5.2. Thus the class C has the MRO $C, A, B,$ `object`.

The C3 algoritm

Python uses the C3 algorithm originally developed for Dylan [12, 2] to find an MRO that satisfies a number of requirements. Most importantly,

- The MRO observes local precedence order. For example, a class A that precedes a class B in the local precedence order of a class C, must precede B in the MRO of C.
- The MRO is monotonic. For example, a class A in the MRO of a class B, must precede B in any MRO that contains A.

To compute the C3 linearization, the inheritance graph is extended with edges for the local precedence order of bases: for each class with ordered list of bases $\overline{C}$ we insert an edge from C_1 to C_2, an edge from C_2 to C_3, and so forth. The ordering of edges is no longer relevant in the extended inheritance graph.

The C3 algorithm produces a depth first topological sort of the extended inheritance graph, provided that the extended graph is acyclic. See for example Figure 5.3.

$$\begin{aligned} &\text{MRO } \Theta(a) = a : \overline{b} \;\sqcup\; \text{MRO } \Theta(b_1) \;\sqcup\; \text{MRO } \Theta(b_2) \;\sqcup\; \cdots \\ &\quad \textbf{where } \overline{b} = \Theta(\Omega_a^{\Theta}(\texttt{\_\_bases\_\_})) \end{aligned} \tag{5.1}$$

The MRO is defined using a left associative merge operator $\overline{a} \;\sqcup\; \overline{b}$ that merges the two topological sorts $\overline{a}$ and $\overline{b}$. Note that, if $\overline{a}$ contains a class that is also in $\overline{b}$, that class will occur only once in $\overline{a} \;\sqcup\; \overline{b}$.

$$\begin{array}{llll} [\,] & \sqcup\ \overline{b} & = \overline{b} & \\ \overline{a} & \sqcup\ [\,] & = \overline{a} & \\ (a:\overline{a}) & \sqcup\ (b:\overline{b}) \text{ if } a \equiv b & = a : \overline{a} & \sqcup\ \overline{b} \\ (a:\overline{a}) & \sqcup\ (b:\overline{b}) \text{ if } a \not\equiv b \wedge a \notin \overline{b} & = a : \overline{a} & \sqcup\ (b:\overline{b}) \\ (a:\overline{a}) & \sqcup\ (b:\overline{b}) \text{ if } a \not\equiv b \wedge b \notin \overline{a} & = b : (a:\overline{a}) & \sqcup\ \overline{b} \end{array} \tag{5.2}$$

Illegal inheritance graphs

Not all inheritance graphs are considered legal, e.g., cyclic extended inheritance graphs are not allowed. This sometimes has unintuitive consequences, see for example Figure 5.4.

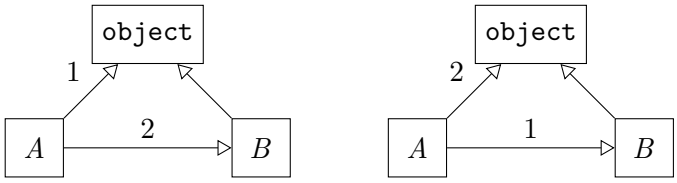

Figure 5.4: The inheritance graph on the right has an illegal ordering of its base classes: the extended inheritance graph has a cycle `object` $\rightarrow B \rightarrow$ `object`. The inheritance graph on the right is legal, because A inherits B before `object` and thus has no cycle in its extended inheritance graph.

Furthermore, all classes must have at least one base, with the exception of `object`, and classes cannot have duplicate bases. Only classes can serve as bases. Finally, a class can only have one built-in superclass (e.g. `int` or `list`).

5.2 Mappings of inherited attributes

As mentioned before, the MRO determines what attributes a class inherits. In order to lookup the attributes of an object Ω_a^Θ, we define two derived mappings that encapsulate the inheritance of attributes.

A mapping of class attributes

We define the derived mapping of class attributes Φ_a^Θ to contain the inherited attributes of the class of a and its own attributes. It is defined using a capital version of the join operator $\oplus$. The capital join operator $\bigoplus_\Theta \overline{a}$ merges the objects $\Omega_{a_1}^\Theta$ through $\Omega_{a_n}^\Theta$, where $n = |\overline{a}|$.

$$\begin{aligned} \textstyle\bigoplus_\Theta [\,] &= \emptyset \\ \textstyle\bigoplus_\Theta a : \overline{a} &= \textstyle\bigoplus_\Theta \overline{a} \oplus \Omega_a^\Theta \end{aligned}$$

The derived mapping Φ_a^Θ is defined as the joined mappings of the MRO of the class of a.

$$\begin{aligned} &\Phi_a^\Theta = \textstyle\bigoplus_\Theta \text{MRO } \Theta(a_c) \\ &\quad \textbf{where } a_c = \Omega_a^\Theta(\texttt{\_\_class\_\_}) \end{aligned} \tag{5.3}$$

A mapping of all attributes

Finally, the full collection of attributes Υ_a^Θ of an object Ω_a^Θ is defined as the combination of its inherited attributes and the object's own attributes.

$$\Upsilon_a^\Theta = \Phi_a^\Theta \oplus \Omega_a^\Theta \tag{5.4}$$

5.3 Classes as types

In the object-oriented world the terms "type" and "classes" are often used interchangeably. In Python, the term "type" is somewhat ambiguously defined. We define two typing relations based on the class-of and base-of relations.

The subclass relation

Classes are said to be subclasses, or subtypes, of their base classes and the base classes thereof. We define the subclass relation $a_a <:_\Theta a_b$ to hold if and only if a_a is a class, and a_b is in the MRO of a_a. If a class a_a is a subclass of a_b, we also say that a_b is a superclass of a_a.

$$a_a <:_\Theta a_b \equiv \texttt{\_\_bases\_\_} \in \Omega^\Theta_{a_a} \wedge a_b \in \text{MRO } \Theta(a_a) \tag{5.5}$$

For example, the class C in Figure 5.3 is a subclass of itself, A, B, and `object`.

The instance-of relation

Before, we described the class-of relation as a relation between object and class: every object has exactly one class. The instance-of relation extends this notion to the superclasses of an object, i.e. an object is an instance of its class and all the superclasses of its class. We define the instance-of relation $a_o :_\Theta a_c$ to hold if and only if the class of a_o is a subclass of a_c.

$$a_o :_\Theta a_c \equiv \Omega^\Theta_{a_o}(\texttt{\_\_class\_\_}) <:_\Theta a_c \tag{5.6}$$

For example, the object c in Figure 5.1 is an instance of C, A, B and `object` (assuming the inheritance graph of Figure 5.2).

6 Basic rewrites rules

6.1 Blocks

Blocks are sequences of statements. Python programs are represented by blocks. Compound statements, such as the while statement and module definitions, contain blocks.

Executing a non-empty block statement

A block is executed statement by statement: the first statement s of the block $s\,;\ b$ is executed, while the remaining statements in block b are put onto the stack for later execution.

$$\begin{array}{l} \left\langle \Theta,\, \Gamma,\, S \quad ,\, s\,;\ b \right\rangle \\ \Rightarrow \left\langle \Theta,\, \Gamma,\, S|b,\, s\,; \quad \right\rangle \end{array} \tag{6.1}$$

Executing an empty block

An empty block is simply popped from the stack, returning the value `None`.

$$\begin{array}{l} \left\langle \Theta,\, \Gamma,\, S,\, \{\} \quad \right\rangle \\ \Rightarrow \left\langle \Theta,\, \Gamma,\, S,\, a_{\texttt{None}} \right\rangle \end{array} \tag{6.2}$$

Rather than creating a new object on the heap, the rule returns a predefined address pointing to the built-in value `None`.

Continuing execution of a block

When the execution the previous statement has completed with result a, the block b on the stack is executed if it is not empty. If b is empty, the block frame is popped and we return a. Thus the result of a block's final statement is the result of the complete block.

$$\begin{array}{l} \left\langle \Theta,\, \Gamma,\, S|b,\, a \right\rangle \\ \Rightarrow \begin{cases} \left\langle \Theta,\, \Gamma,\, S \quad ,\, a \right\rangle & \text{if } b = \{\} \\ \left\langle \Theta,\, \Gamma,\, S \quad ,\, b \right\rangle & \text{otherwise} \end{cases} \end{array} \tag{6.3}$$

6.2 Pass and expression statements

Pass statements

The pass statement affects neither the heap, the environment nor the stack and returns `None`.

$$\begin{aligned} &\langle \Theta, \Gamma, S, \texttt{pass}; \rangle \\ \Rightarrow &\langle \Theta, \Gamma, S, a_{\texttt{None}} \rangle \end{aligned} \tag{6.4}$$

Expression statements

Expressions also serve as statements. Syntactically these expression statements are indistinguishable from normal expressions, therefore they are annotated with a suffix ;

$$\begin{aligned} &\langle \Theta, \Gamma, S, e; \rangle \\ \Rightarrow &\langle \Theta, \Gamma, S, e \rangle \end{aligned} \tag{6.5}$$

7 Variables

Variables are references to values on the heap. Variables can be assigned, evaluated (dereferenced), and deleted. Because variables are references, assignments and deletions change only the variable binding, not the values themselves. Multiple values can also refer to the same value. The program in Listing 7.1 shows how a references can be used to share values.

```
x = [1,2]   # assigning a variable with a list
y = x       # aliasing the variable
x.append(3) # append 3 to the list
print(y)    # prints [1,2,3]
x = []      # reassigning x to the empty list
print(y)    # prints [1,2,3]
z = y
del y       # removes the binding for y
print(z)    # prints [1,2,3]
```

Listing 7.1: Assignments change the variable but not the value

7.1 Assignment statements

Executing an assignment statement

Executing the assignment $x = e$ starts with the evaluation of e. A continuation frame $x = \circ$, holding the variable name x, is placed on the stack.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S \quad\quad\quad ,\, x = e; \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|x = \circ,\, e \quad\quad \rangle \end{array} \tag{7.1}$$

The circle $\circ$ in the stack frame $x = \circ$ serves as a placeholder for the expression e that is being evaluated.

Binding a variable

When the evaluation of e is completed, the resulting address a is stored in the topmost environment mapping. Any previous binding of x in the mapping $\Theta(\gamma_1)$ will be overwritten.

$$\begin{array}{l} \langle \Theta \quad\quad\quad\quad\quad\quad\quad\quad\quad ,\, \Gamma|\gamma_1,\, S|x = \circ,\, a \quad\quad \rangle \\ \Rightarrow \langle \Theta \oplus [\gamma_1 \rightarrow \Theta(\gamma_1) \oplus [x \rightarrow a]],\, \Gamma|\gamma_1,\, S \quad\quad\quad ,\, a_{\texttt{None}} \rangle \end{array} \tag{7.2}$$

7.2 Variable expressions

Evaluating a variable expression

Evaluating a variable in general amounts to a simple lookup in the environment: if the variable is bound in the environment, we return it, otherwise an exception is raised.

Local variables (see Section 8.6), have slightly different semantics. A local variable can only be retrieved from the local environment, which is topmost in the environment stack.

Thus, the evaluation of a variable has four distinct cases:

1. If x is a local variable and is bound in the local environment $\Theta(\gamma_1)$, we lookup and return the address of x.
2. An `UnboundLocalError` exception is raised if x is a local variable but is unbound in the local environment.
3. If x not a local variable and is bound in the full environment $\Sigma^{\Theta}_{\Gamma|\gamma_1}$, we lookup and return the address of x.
4. Otherwise the variable is unbound and not local, so we raise a `NameError`.

$$
\begin{array}{l}
\quad \langle \Theta,\, \Gamma|\gamma_1,\, S,\, x \qquad\qquad\qquad\qquad\quad \rangle \\
\Rightarrow \begin{cases}
\langle \Theta,\, \Gamma|\gamma_1,\, S,\, \Theta(\gamma_1)(x) \qquad\qquad\quad \rangle & \text{if } x \in \Theta(\gamma_1)_{loc} \wedge x \in \Theta(\gamma_1) \\
\langle \Theta,\, \Gamma|\gamma_1,\, S,\, \texttt{raise } a_{\texttt{UnboundLocalError}}() \rangle & \text{if } x \in \Theta(\gamma_1)_{loc} \wedge x \notin \Theta(\gamma_1) \\
\langle \Theta,\, \Gamma|\gamma_1,\, S,\, \Sigma^{\Theta}_{\Gamma|\gamma_1}(x) \qquad\quad \rangle & \text{if } x \notin \Theta(\gamma_1)_{loc} \wedge x \in \Sigma^{\Theta}_{\Gamma|\gamma_1} \\
\langle \Theta,\, \Gamma|\gamma_1,\, S,\, \texttt{raise } a_{\texttt{NameError}}() \qquad\quad \rangle & \text{otherwise}
\end{cases}
\end{array}
\tag{7.3}
$$

7.3 Delete statements

Executing a variable deletion

The delete statement `del` x removes the variable x from the local environment γ_1. As for the variable evaluation, local variables have a special case:

1. If x is bound in the local environment, we delete it.
2. An `UnboundLocalError` is raised if x is a local variable and not (yet) bound in the local environment.
3. Otherwise the variable must be unbound and not local, so we raise a `NameError`.

$$
\begin{array}{l}
\quad \langle \Theta\ ,\, \Gamma|\gamma_1,\, S,\, \texttt{del } x; \qquad\qquad\quad \rangle \\
\Rightarrow \begin{cases}
\langle \Theta',\, \Gamma|\gamma_1,\, S,\, a_{\texttt{None}} \qquad\qquad\quad \rangle & \text{if } x \in \Theta(\gamma_1) \\
\langle \Theta\ ,\, \Gamma|\gamma_1,\, S,\, \texttt{raise } a_{\texttt{UnboundLocalError}}() \rangle & \text{if } x \notin \Theta(\gamma_1) \wedge x \in \Theta(\gamma_1)_{loc} \\
\langle \Theta\ ,\, \Gamma|\gamma_1,\, S,\, \texttt{raise } a_{\texttt{NameError}}() \qquad\quad \rangle & \text{otherwise}
\end{cases} \\
\textbf{where } \Theta' = \Theta \oplus [\gamma_1 \to \Theta(\gamma_1) \ominus x]
\end{array}
\tag{7.4}
$$

8 Functions and generators

This chapter discusses functions and generators. Following the lifetime of a function, we define the semantics of functions in three steps: first the function definition, followed by the function call, execution, and the return. Next we discuss the operation of generators: (re)starting a generator, and 'returning' from a generator with the yield statement.

Functions in Python behave very similar to functions in other imperative language. A function takes a number of arguments, executes its body, and returns some value. For example Listing 8.1 shows how a factorial function is defined and called.

```
def fac(x) :
       if x < 1 :
              return 1
       else :
              return x * fac(x-1)

fac(5) # returns 120
```

Listing 8.1: A recursive factorial function

Generators are a unique concept of Python. They are defined using a normal function definition that contains one or more yield statements and no return statements (see Chapter 3). Such a generator function returns a generator when called.

When a generator's primitive function `__next__` is called, the generator executes the body of the generator function until a yield expression `yield` e is evaluated. Evaluating the yield expression causes the generator to suspend execution and return e. When `__next__` is called again, execution continues at the point where it left off, until either another yield expression is encountered or the end of the body is reached. At the end of its body, a generator stops and raises an exception.

For example Listing 8.2 shows how a generator is used to create a generator that returns the elements of a list one by one.

8.1 Defining functions and generators

A function definition `def` $x_\lambda(\overline{x}) : b_\lambda$ consists of: the function's name x_λ, the list of parameter names $\overline{x}$, and the body b_λ. When executed, a new function object is added to the heap, the function name is bound in the environment, and `None` is returned.

Functions which body contains a yield expression, produce generators when called (see Section 8.3). The definition of these generator functions however, is no different from normal functions.

```
def createListGenerator(list) :
       try :
              i = 0
              while True :
                     yield list[i]
                     i = i + 1
       except IndexError, e :     # catch index out of bounds
              pass

it = createListGenerator([1,2])
it.next() # returns 1
it.next() # returns 2
it.next() # raises a StopIteration exception
```

Listing 8.2: A generator function that produces a list iterator

Executing a function definition statement

The new function object stored at a_λ consists of two parts: the map of attributes $[\texttt{\_\_class\_\_} \to a_{\texttt{function}}]$, which only contains the class attribute, and a function closure value. The closure $\lambda\overline{x}.\ b_\lambda \dashv \Gamma_\lambda$ consists of: the function arguments $\overline{x}$, its body b_λ, and its environment Γ_λ.

$$\begin{aligned}
&\langle \Theta\ ,\ \Gamma|\gamma_1,\ S,\ \texttt{def}\ x_\lambda(\overline{x}) : b_\lambda; \rangle \\
\Rightarrow\ &\langle \Theta',\ \Gamma|\gamma_1,\ S,\ a_{\texttt{None}} \rangle \\
&\textbf{where}\ a_\lambda = \text{new address} \notin \Theta \\
&\qquad \Gamma_\lambda = \textbf{if}\ \Theta(\gamma_1) \in \Omega\ \textbf{then}\ \Gamma\ \textbf{else}\ \Gamma|\gamma_1 \\
&\qquad \Theta' = \Theta \oplus [\, a_\lambda \to ([\texttt{__class__} \to a_{\texttt{function}}], \lambda\overline{x}.\ b_\lambda \dashv \Gamma_\lambda) \\
&\qquad\qquad , \gamma_1 \to \Theta(\gamma_1) \oplus [x_\lambda \to a_\lambda] \\
&\qquad\qquad]
\end{aligned} \tag{8.1}$$

Class functions, recognized by the topmost environment (see Section 9.1), cannot access other class members directly. Therefore, the environment Γ_λ of class functions does not include the local environment γ_1 of the function definition statement. See Example 8.3.

```
x = 1
class A(object) :
       x = x*2 # define class variable A.x

       def foo(self) :
              return x

print(A().foo()) # prints 1
print(A.x)      # prints 2
```

Listing 8.3: Class variables are not in the scope of class variables

8.2 Executing functions

Evaluating calls

A call expression $e_f(\overline{e})$ executes a function e_f with arguments $\overline{e}$. Before the function is executed, e_f and $\overline{e}$ are evaluated from left to right.

Evaluating the function expression

First, the function expression e_f is evaluated. A frame $\circ(\overline{e})$ is pushed onto the stack containing the remaining argument expressions.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S \qquad ,\, e_f(\overline{e}) \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|\circ(\overline{e}),\, e_f \quad \rangle \end{array} \tag{8.2}$$

Evaluating the first argument expression

Then, when e_f has been evaluated to a_f, we continue to evaluate the first argument e'_1 and push the remaining arguments $\overline{e}'$ and a_f on the stack. If there are no arguments, a_f is executed (see Section 8.5).

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S|\circ(\overline{e}) \quad ,\, a_f \quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S|a_f(\circ,\overline{e}'),\, e \quad \rangle & \text{if } \overline{e} = (e:\overline{e}') \\ \langle \Theta,\, \Gamma,\, S \qquad ,\, a_f() \rangle & \text{otherwise} \end{cases} \end{array} \tag{8.3}$$

Evaluating the remaining argument expressions

Finally, the remaining arguments $\overline{e}$ on the stack are evaluated one by one. When all argument expressions have been evaluated, the function is executed.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S|a_f(\overline{a},\circ,\overline{e}) \qquad ,\, a \qquad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S|a_f(\overline{a} +\!\!+ [a],\circ,\overline{e}'),\, e \qquad \rangle & \text{if } \overline{e} = (e:\overline{e}') \\ \langle \Theta,\, \Gamma,\, S \qquad ,\, a_f(\overline{a} +\!\!+ [a]) \rangle & \text{otherwise} \end{cases} \end{array} \tag{8.4}$$

Type checking an annotated function

The instance-of relation is not very informative for functions, since all functions are instances of the class `function`. However, we can describe a function more precisely by the types of arguments it accepts.

Function annotations describe the type of a function's arguments, which are checked right before execution of the function. For example the primitive addition function (see Rule A.6) is annotated (see Section A.3) to accept only instances of `int` as arguments.

Annotations are only introduced by rewrite rules and object definitions in these semantics. There is no annotation in the abstract syntax.

Type checking function arguments

An annotated function $a_f : \overline{c}$ consists of a function address a_f and a list of classes $\overline{c}$. We execute the annotated function call with arguments $\overline{a}$, if all arguments $\overline{a}$ are instances of classes $\overline{c}$. There may be fewer classes in $\overline{c}$ than there are arguments, but not more. If the arguments fail to satisfy the type, we raise an exception (see Chapter 10).

$$\begin{array}{l} \left\langle \Theta,\, \Gamma,\, S,\, a_f : \overline{c}(\overline{a}) \right\rangle \\ \Rightarrow \begin{cases} \left\langle \Theta,\, \Gamma,\, S,\, a_f(\overline{a}) \right\rangle & \text{if } |\overline{a}| \geqslant |\overline{c}| \wedge \forall_{0<i<|\overline{c}|}\ \overline{a}_i :_\Theta \overline{c}_i \\ \left\langle \Theta,\, \Gamma,\, S,\, \texttt{raise}\ a_{\texttt{TypeError}}() \right\rangle & \text{otherwise} \end{cases} \end{array} \tag{8.5}$$

Function closure execution

The execution of a fully evaluated call $a_\lambda(\overline{a})$ has four (successful) cases: a_λ is a function closure, a_λ is a generator function, a_λ is a callable object, or a_λ is a primitive function. The latter case is not handled in this chapter. Instead, every primitive function has a specific rewrite rule. See for example the primitive addition function which is handled in Rule A.6.

Execution of a function

The execution of a function call has four cases:

1. If a_f is a instance of `function` then the value of a_f is the closure $\lambda\overline{x}.\ b_\lambda \dashv \Gamma_\lambda$. When the number of arguments $\overline{a}$ equals the number of parameters $\overline{x}$, and a_f is not a generator function, we execute the function body b_λ with the environment $\Gamma_\lambda|\gamma_1$. The current environment Γ is pushed on the stack in the return marker $\Gamma \vdash \circ_\lambda$.

2. If a_f is a closure with the correct number of arguments, and a_f is a generator function, we return a generator $\langle b_\lambda, \Gamma_\lambda|\gamma_1\rangle$ (see Section 8.3). The generator's environment is the same as a function's environment would be, and the only element on its stack is the function body.

3. If a_f is not a closure, but an object with a class attribute `__call__`, we forward the call to that attribute. The function is instantiated with the object a_f.

4. Otherwise, the call had an incorrect number of arguments or a_f was neither a function nor a callable object, so we raise a `TypeError`.

$$
\begin{array}{l}
\quad \langle \Theta \ , \Gamma \quad\ , S \quad\ , a_f(\overline{a}) \quad\ \rangle \\
\Rightarrow \begin{cases}
\langle \Theta' \, , \Gamma_\lambda | \gamma_1, \, S | \Gamma \vdash \circ_\lambda, \, b_\lambda \ \rangle & \text{if } a_f :_\Theta a_{\texttt{function}} \quad \wedge |\overline{a}| \equiv |\overline{x}| \wedge \texttt{yield} \notin b_\lambda \\
\langle \Theta'', \Gamma \quad\ , S \quad\ , a_{\text{gen}} \ \rangle & \text{if } a_f :_\Theta a_{\texttt{function}} \quad \wedge |\overline{a}| \equiv |\overline{x}| \wedge \texttt{yield} \in b_\lambda \\
\langle \Theta \ , \Gamma \quad\ , S \quad\ , a_f.a_{call}(\overline{a}) \ \rangle & \text{if } \neg\,(a_f :_\Theta a_{\texttt{function}}) \wedge \texttt{\_\_call\_\_} \in \Phi^\Theta_{a_f} \\
\langle \Theta \ , \Gamma \quad\ , S \quad\ , \texttt{raise } a_e() \rangle & \text{otherwise}
\end{cases} \\
\quad \textbf{where } \gamma_1 \ = \text{new address} \notin \Theta \\
\qquad\quad \Theta' \ = \Theta \oplus [\gamma_1 \ \ \rightarrow ([\overline{x}_1 \rightarrow \overline{a}_1, \cdots, \overline{x}_n \rightarrow \overline{a}_n], \mathrm{lv}(b_\lambda) \cup \overline{x})] \\
\qquad\quad a_{\text{gen}} = \text{new address} \notin \Theta' \\
\qquad\quad \Theta'' \ = \Theta' \oplus [\, a_{\text{gen}} \rightarrow \langle b_\lambda, \Gamma_\lambda | \gamma_1 \rangle] \\
\qquad\quad a_{call} = \Phi^\Theta_{a_f}(\texttt{\_\_call\_\_}) \\
\qquad\quad a_e \ \ = a_{\texttt{TypeError}} \\
\qquad\quad \lambda \overline{x}.\ b_\lambda \dashv \Gamma_\lambda = \Theta(a_f)
\end{array}
\tag{8.6}
$$

The function's local environment γ_1 has two parts: the parameter-argument bindings $[\overline{x}_1 \rightarrow \overline{a}_1, \cdots, \overline{x}_n \rightarrow \overline{a}_n]$ and the set of local variables $\mathrm{lv}(b_\lambda) \cup \overline{x}$.

The local variables of a block are the variables used as a target in body of the function, i.e., the variable that can be bound in the local environment when the body is executed. Variable expressions and delete statements have different semantics for local variables (see Chapter 7).

Local variables

The set of local variables $\mathrm{lv}(b)$ consists of all the variables that can be (un)bound during the execution a block b. Apart from the assign and delete statements, the for and try statements can also bind variables. There is no expression that binds variables. In that respect they are truly stateless, in contrast to statements.

$$
\begin{array}{lll}
\mathrm{lv}(\{\}) & = \emptyset \\
\mathrm{lv}(x = e\texttt{;} & b) = x \cup \mathrm{lv}(b) \\
\mathrm{lv}(\texttt{del } x\texttt{;} & b) = x \cup \mathrm{lv}(b) \\
\mathrm{lv}(\texttt{while } e : b_x\texttt{;} & b) = \quad \mathrm{lv}(b) \cup \mathrm{lv}(b_x) \\
\mathrm{lv}(\texttt{for } x \texttt{ in } e : b_x\texttt{;} & b) = x \cup \mathrm{lv}(b) \cup \mathrm{lv}(b_x) \\
\mathrm{lv}(\texttt{if } e : b_x \texttt{ else} : b_y\texttt{;} & b) = \quad \mathrm{lv}(b) \cup \mathrm{lv}(b_x) \cup \mathrm{lv}(b_y) \\
\mathrm{lv}(\texttt{try} : b_x \texttt{ except } e\texttt{, } x : b_y\texttt{;} & b) = x \cup \mathrm{lv}(b) \cup \mathrm{lv}(b_x) \cup \mathrm{lv}(b_y) \\
\mathrm{lv}(\texttt{try} : b_x \texttt{ finally} : b_y\texttt{;} & b) = \quad \mathrm{lv}(b) \cup \mathrm{lv}(b_x) \cup \mathrm{lv}(b_y) \\
\mathrm{lv}(s\texttt{;} & b) = \quad \mathrm{lv}(b)
\end{array}
$$

Returning from a function

A function stops execution at the end of its block or when a return statement is encountered. When a function returns, it returns a value and the environment stored on the stack is reinstated. Functions always return a value; those without a return statement return `None`.

Returning from the end of a function

When all statements in b_λ have been executed, but no return statement has been executed, we will encounter the return marker $\Gamma \vdash \circ_\lambda$. The return marker contains the old environment Γ

which is reinstated. Since there was no return statement, we return `None`.

$$\begin{array}{l} \langle \Theta,\, \Gamma_\lambda,\, S|\Gamma \vdash \circ_\lambda,\, a \rangle \\ \Rightarrow \langle \Theta,\, \Gamma\ ,\, S\ ,\, a_{\texttt{None}} \rangle \end{array} \tag{8.7}$$

Executing a return statement

At a return statement, the returned expression e is evaluated and a return frame `return` $\circ$ is placed on the stack.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S\ ,\, \texttt{return}\ e; \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|\texttt{return}\ \circ\,,\, e\ \rangle \end{array} \tag{8.8}$$

Start stack unwinding

Next, once the returned value e has been evaluated to a, we use the return instruction to unwind the stack and return a

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S|\texttt{return}\ \circ\,,\, a\ \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S\ ,\, \texttt{return}\ a \rangle \end{array} \tag{8.9}$$

Unwinding the stack

Finally the return instruction unwinds (pops frames from) the stack until the return frame is found on top of the stack. At the return frame the old environment is reinstated, a is returned, and unwinding stops.

$$\begin{array}{l} \langle \Theta,\, \Gamma\ ,\, S|f\ ,\, \texttt{return}\ a \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma',\, S\ ,\, a\ \rangle & \text{if } f = \Gamma' \vdash \circ_\lambda \\ \langle \Theta,\, \Gamma\ ,\, S|\texttt{return}\ a,\, b\ \rangle & \text{if } f = \texttt{finally} : b \\ \langle \Theta,\, \Gamma\ ,\, S\ ,\, \texttt{return}\ a \rangle & \text{otherwise} \end{cases} \end{array} \tag{8.10}$$

If the return statement is in the body of a try-finally statement, we execute the finally clause `finally` : b before returning from the function.

8.3 Executing generators

In the previous sections we have already seen how a generator function is defined and executed. In this section we describe how generators are executed.

The execution of a generator function returned a generator $\langle S, \Gamma \rangle$ with a stack S and environment Γ. When the generator itself is executed, the stack S and environment Γ become the stack and environment of the abstract machine. The generator is suspended by storing the stack and environment in the generator value, and reinstating the old stack and environment.

The following rules describe how the generator is (re)started using the primitive functions `__next__` or `__send__`, and how it is suspended by the yield expression.

Getting the next yielded value

The primitive function `__next__` starts or resumes a generator if it hasn't already started.

If `__next__` is called for generator that has already started, i.e., it has a value of $\langle\rangle$, we simply raise a `ValueError`. A running generator cannot be (re)started.

If `__next__` is called for a suspended or new generator $\langle S_{\text{gen}}, \Gamma_{\text{gen}} \rangle$ the stack S_{gen} is pushed on the machine stack and the environment is set to Γ_{gen}. A yield marker $\Gamma \vdash a_{\texttt{self}}$ with the current environment Γ and the generator's address $a_{\texttt{self}}$ is pushed on the stack, before S_{gen} the generator's stack.

$$
\begin{array}{l}
\quad \langle \Theta \;,\; \Gamma \quad,\; S \qquad\qquad\qquad\quad ,\; \texttt{generator.\_\_next\_\_}([\, a_{\texttt{self}} \,]) \rangle \\
\Rightarrow \begin{cases} \langle \Theta', \Gamma_{\text{gen}},\; S | \Gamma \vdash a_{\texttt{self}} | S_{\text{gen}},\; a_{\texttt{None}} \qquad\qquad\quad \rangle & \text{if } v = \langle S_{\text{gen}}, \Gamma_{\text{gen}} \rangle \\ \langle \Theta \;,\; \Gamma \quad,\; S \qquad\qquad\qquad\quad ,\; \texttt{raise } a_{\texttt{ValueError}}() \quad \rangle & \text{if } v = \langle\rangle \end{cases} \\
\quad \textbf{where } \Theta' = \Theta \oplus [\, a_{\texttt{self}} \to \langle\rangle \,] \\
\qquad\quad v \;\; = \Theta(a_{\texttt{self}})
\end{array}
\tag{8.11}
$$

Getting the next value with yield value

The primitive `__send__` resumes a generator like `__next__`, but there is one difference: calling `__send__` with argument a causes the yield expression where the generator has suspended, to evaluate to a.

$$
\begin{array}{l}
\quad \langle \Theta \;,\; \Gamma \quad,\; S \qquad\qquad\qquad\quad ,\; \texttt{generator.\_\_send\_\_}([\, a_{\texttt{self}}, a \,]) \rangle \\
\Rightarrow \begin{cases} \langle \Theta', \Gamma_{\text{gen}},\; S | \Gamma \vdash a_{\texttt{self}} | S_{\text{gen}},\; a \qquad\qquad\qquad\quad \rangle & \text{if } v = \langle S_{\text{gen}}, \Gamma_{\text{gen}} \rangle \\ \langle \Theta \;,\; \Gamma \quad,\; S \qquad\qquad\qquad\quad ,\; \texttt{raise } a_{\texttt{ValueError}}() \quad \rangle & \text{if } v = \langle\rangle \end{cases} \\
\quad \textbf{where } \Theta' = \Theta \oplus [\, a_{\texttt{self}} \to \langle\rangle \,] \\
\qquad\quad v \;\; = \Theta(a_{\texttt{self}})
\end{array}
\tag{8.12}
$$

The yield expression

Evaluating a yield expression

First, the value e to be returned is evaluated.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S \qquad\qquad ,\, \texttt{yield}\ e \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|\texttt{yield}\ \circ\,,\, e \qquad \rangle \end{array} \tag{8.13}$$

Start unwinding of the stack

Then, with the yielded value a, we start to unwind the stack.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S|\texttt{yield}\ \circ\,,\, a \qquad\quad \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S \qquad\qquad ,\, \texttt{yield}\ a,\, \square \rangle \end{array} \tag{8.14}$$

The second part of the yield instruction $\texttt{yield}\ a,\, \square$ stores the part of the stack that has been unwound. Unwinding starts with an empty stack ($\square$).

Unwinding the stack

Unwinding of the stack continues until a yield marker $\gamma \vdash a_{\text{gen}}$ is on top. At that point, the generator a_{gen} is updated with the unwound stack S_{gen} and the generator's environment Γ_{gen}. Thus the state of the generator at the yield statement is stored.

$$\begin{array}{l} \langle \Theta \qquad\qquad\qquad\qquad\qquad\qquad ,\, \Gamma_{\text{gen}},\, S|f,\, \texttt{yield}\ a,\, S_{\text{gen}} \rangle \\ \Rightarrow \begin{cases} \langle \Theta \oplus [\, a_{\text{gen}} \to \langle S_{\text{gen}}, \Gamma_{\text{gen}} \rangle], \Gamma \quad ,\, S \quad ,\, a \qquad\qquad\quad \rangle & \text{if } f = \Gamma \vdash a_{\text{gen}} \\ \langle \Theta \qquad\qquad\qquad\qquad\qquad\qquad ,\, \Gamma_{\text{gen}},\, S \quad ,\, \texttt{yield}\ a,\, f|S_{\text{gen}} \rangle & \text{otherwise} \end{cases} \end{array} \tag{8.15}$$

Stopping a generator

A generator stops when it reaches the end of its body, or an exception is raised in its body (see Rule 10.5). Stopped generators cannot be restarted, they will raise a `StopIteration` exception when calling `__next__`.

Stopping at the end of a generator

When the yield marker $\Gamma \vdash a_{\text{gen}}$ is on top of the stack, the generator's body has been executed. We stop the generator, raise a `StopIteration` and reinstate the old environment.

$$\begin{array}{l} \langle \Theta \qquad\qquad\qquad\quad ,\, \Gamma_{\text{gen}},\, S|\Gamma \vdash a_{\text{gen}},\, a \qquad\qquad\qquad\quad \rangle \\ \Rightarrow \langle \Theta \oplus [\, a_{\text{gen}} \to \langle \square, \square \rangle], \Gamma \quad ,\, S \qquad\qquad ,\, \texttt{raise}\ a_{\texttt{StopIteration}}() \rangle \end{array} \tag{8.16}$$

The generator stored on the heap at address a_{gen} is updated to the empty generator $\langle \square, \square \rangle$. This will cause subsequent executions of the generator to immediately stop.

9 Classes and objects

A number of statements and expressions are devoted to classes and objects. Most of which are common to object-oriented language. Class definitions, attribute access, attribute assignments and class functions are all common programming constructs.

Python differs from most other languages in its extremely dynamic implementation of objects: Python's objects, and even classes, can be altered at any time. Attributes, including the `__class__` and `__bases__` attributes, can be removed and (re)defined using delete and assignment statements. Furthermore, the semantics of attribute references, assignments, and deletions can be changed by overriding class members.

The example in Listing 9.1 shows how the semantics of attribute assignment are overridden by the class function `__setattr__`, and how the attribute deletion statement removes `__setattr__` even after the class has been created.

```
class Counter(object) :
       def __init__(self) :
              object.__setattr__(self, "x", 0)
       def inc(self) :
              object.__setattr__(self, "x", self.x + 1)
       def __setattr__(self, attr, val) :
              print("Use␣inc()␣to␣change␣the␣counter␣value")


c = Counter()
print(c.x)             # prints 0
c.inc()
print(c.x)             # prints 1
c.x = 0                # prints the error message
print(c.x)             # prints 1

del Counter.__setattr__ # remove the class function __setattr__
c.x = 5                # nothing is printed
print(c.x)             # prints 5
```

Listing 9.1: A simple use-case for classes and objects. The Counter class overrides `__setattr__` to prevent direct access to the internal counter x. Because the class overrides `__setattr__` we are forced to use `object.__setattr__` to set the attributes of a counter.

9.1 Class definition

A class definition `class` $x(e) : b$ creates a new class object. Its bases are set to the value of e, and the variables bound in b define its attributes.

Class definitions are not declarative, like in many other object oriented languages, but imperative in style. The body b and bases e of a new class are executed and evaluated when the class definition is executed.

Evaluating the bases of a new class

Before the body is executed, the base objects have to be evaluated. The bases are represented by a tuple expression e.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S \qquad\qquad\qquad\quad ,\, \texttt{class}\ x(e) : b; \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|\texttt{class}\ x(\circ) : b,\, e \qquad\qquad\quad \rangle \end{array} \tag{9.1}$$

Although one would expect to derive only from instances of `type`, any object is accepted as a base at this point.

Executing the body of a class definition

When e has been evaluated, its result a_b is stored on the stack.

The body is executed with a new object γ_c on top of the environment stack. All variable bindings within the body will be bound in the object γ_c. Thus attributes of the new class are defined by ordinary variable bindings in the body.

$$\begin{array}{l} \quad \langle \Theta \qquad\qquad\qquad\quad ,\, \Gamma|\gamma_1,\, S|\texttt{class}\ x(\circ) : b \qquad\quad ,\, a_b \rangle \\ \Rightarrow \langle \Theta \oplus [\gamma_c \to (\emptyset, \texttt{None})],\, \Gamma' \quad ,\, S|\Gamma|\gamma_1 \vdash \texttt{class}\ x(a_b) : \circ,\, b \ \rangle \\ \textbf{where}\ \gamma_c = \text{new address} \notin \Theta \\ \qquad \Gamma' = \textbf{if}\ \Theta(\gamma_1) \text{ is object } \textbf{then}\ \Gamma|\gamma_c \\ \qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \Gamma|\gamma_1|\gamma_c \end{array} \tag{9.2}$$

Class definitions that are nested in class definitions, can not access the attributes of the class enclosing them through the environment. If the topmost environment γ_1 is an object, the body of the class definition is executed without γ_1 in the environment stack. Nested function definitions are limited in the same way (see Equation 8.1).

Creating a new class

After the body has been executed, we store the class name x on the heap and construct a new class object. The new object is constructed by the constructor of the metaclass i.e., the class function `__new__`. See Rule 9.4 for the default constructor.

$$\begin{array}{l} \quad \langle \Theta\ ,\, \Gamma'|\gamma_c,\, S|\Gamma|\gamma_1 \vdash \texttt{class}\ x(a_b) : \circ\ ,\, a \qquad\qquad\qquad\quad \rangle \\ \Rightarrow \langle \Theta',\, \Gamma\ |\gamma_1,\, S|\ \circ_x .\texttt{\_\_init\_\_}(a_x, a_b, \gamma_c),\, \Phi^{\Theta}_{a_t}(\texttt{\_\_new\_\_})([\,a_t, a_x, a_b, \gamma_c]) \rangle \\ \textbf{where}\ \Theta' = \Theta \oplus [\,a_x \to x\,] \\ \qquad a_x = \text{new address} \notin \Theta \\ \qquad a_t\ = \textbf{if}\ \texttt{\_\_metaclass\_\_} \in \Theta(\gamma_c)\ \textbf{then}\ \Theta(\gamma_c)(\texttt{\_\_metaclass\_\_}) \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ a_{\texttt{type}} \end{array} \tag{9.3}$$

Note that this implementation of metaclasses is incomplete: in Python the metaclass is inherited from the base(s) of a class. This is more complicated than it seems at first glance. For example, when a new class C is created that inherits from two classes A and B which inherit from two different metaclasses M_A and M_B respectively, it is unclear which metaclass constructs C. Python will raise an exception in this particular situation. For the moment *minpy* avoids these complications by disabling inheritance of metaclasses.

The default class constructor

The default class constructor extends the new class a_c with name a_x, bases a_b and class $a_{\texttt{self}}$. The `__metaclass__` attribute, which typically refers to $a_{\texttt{self}}$, is removed from the new class. If no valid inheritance graph can be created because the bases are bad (see 5.1), we raise a `TypeError`.

$$
\begin{aligned}
&\left\langle \Theta\ ,\ \Gamma,\ S,\ \texttt{type.\_\_new\_\_}([\,a_{\texttt{self}}, a_x, a_b, a_c\,])\right\rangle \\
\Rightarrow &\begin{cases} \left\langle \Theta',\ \Gamma,\ S,\ a_c \right\rangle & \text{if } \Theta'(a_c) \text{ is valid} \\ \left\langle \Theta\ ,\ \Gamma,\ S,\ \texttt{raise } a_{\texttt{TypeError}}() \right\rangle & \text{otherwise} \end{cases} \\
&\textbf{where } \Theta' = \Theta \oplus [\,a_c \rightarrow (\Theta(a_c) \oplus [\,\texttt{\_\_class\_\_} \rightarrow a_{\texttt{self}} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ,\texttt{\_\_bases\_\_} \rightarrow a_b \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad]) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ominus \texttt{\_\_metaclass\_\_} \\
&\qquad\qquad\qquad\qquad\qquad\qquad]
\end{aligned}
\tag{9.4}
$$

Initializing the new class

New classes, created by the constructor `__new__` of the metaclass, are initialized by the `__init__` function of the metaclass. So when the constructor returns, the initializer is called.

The assignment on the stack binds the class name x to the new class a_o. If the initialization function raises an exception, the assignment will be popped from the stack and the class will not be bound.

$$
\begin{aligned}
&\left\langle \Theta,\ \Gamma,\ S|\circ_x .\texttt{\_\_init\_\_}(a_x, a_b, \gamma_c),\ a_o \right\rangle \\
\Rightarrow &\left\langle \Theta,\ \Gamma,\ S|x = a_o;\ ,\ \Phi^{\Theta}_{a_t}(\texttt{\_\_init\_\_})([\,a_t, a_x, a_b, \gamma_c\,])\right\rangle \\
&\textbf{where } a_t = \Theta(a_o)(\texttt{\_\_class\_\_})
\end{aligned}
\tag{9.5}
$$

The original metaclass, whose constructor was called, can return any object. Therefore we call the initialization function of the returned object's class, rather than the original metaclass.

9.2 Object creation

The creation of objects is handled by the `__call__`, `__new__`, and `__init__` attributes. Most classes will inherit those attributes from the base object `object`, whose primitives define the default construction mechanism.

The default constructor

Every class can be called as if it were a function because all classes inherit, or override, this `__call__` primitive. If a class is called, its `__call__` attribute function is called (see Rule 8.6). The `__call__` primitive consecutively calls the `__new__` and `__init__` to construct a new class and initialize it.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S \qquad\qquad\qquad\quad ,\, \texttt{type.\_\_call\_\_}(a_c : \overline{a}) \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|\circ.\texttt{\_\_init\_\_}(\overline{a}),\, a_f(a_c : \overline{a}) \qquad\quad \rangle \\ \textbf{where } a_f = (\bigoplus_\Theta \text{MRO } \Theta(a_c))(\texttt{\_\_new\_\_}) \end{array} \tag{9.6}$$

The `__new__` function a_f, is looked up in the class itself or its bases. The frame $\circ.\texttt{\_\_init\_\_}(\overline{a})$, is put onto the stack to execute the `__init__` with the same arguments.

The default object creation

The default implementation of `__new__` in `object` creates a new object with base `None`, and sets its class to a_c.

$$\begin{array}{l} \langle \Theta \qquad\qquad\qquad\qquad\qquad\qquad\quad ,\, \Gamma,\, S,\, \texttt{object.\_\_new\_\_}(a_c : \overline{a}) \rangle \\ \Rightarrow \langle \Theta \oplus [\,a \to ([\texttt{\_\_class\_\_} \to a_c], \texttt{None})],\, \Gamma,\, S,\, a \qquad\qquad\quad \rangle \\ \textbf{where } a = \text{new address} \notin \Theta \end{array} \tag{9.7}$$

Primitive values that allow subclassing, have their own `__new__` attributes which initialize the base value to their primitive value. See for example Rule A.8.

From creation to initialization

The new object a returned by `__new__` is initialized by its class. If the initialization function does not raise an exception, the new object a on the stack will be returned.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S|\circ.\texttt{\_\_init\_\_}(\overline{a}),\, a \qquad\quad \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|a \qquad\qquad ,\, a_f(a : \overline{a}) \rangle \\ \textbf{where } a_f = \Phi_a^\Theta(\texttt{\_\_init\_\_}) \end{array} \tag{9.8}$$

The default object initialization

The default initialization does nothing at all and returns `None`.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S,\, \texttt{object.\_\_init\_\_}(\overline{a}) \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S,\, a_{\texttt{None}} \qquad\qquad\quad \rangle \end{array} \tag{9.9}$$

Like `object.__new__`, this primitive accepts an arbitrary number of arguments. This allows a user defined `__new__` function to accept any number of arguments.

9.3 Attribute access

The attribute access expression $e.x$ evaluates to the attribute x of object e. However, its semantics can be changed by overriding the `__getattribute__` function.

Evaluating the attribute access expression

First the left hand side expression e of the expression $e.x$ is evaluated.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S \quad\quad ,\, e.x \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S| \circ .x,\, e \quad \rangle \end{array} \tag{9.10}$$

Calling `__getattribute__`

When the left hand side expression e has been evaluated to an object a, the `__getattribute__` function of a is called. The function `__getattribute__` is called with the evaluted object a (we instantiate `__getattribute__` with a) and the attribute name a_x.

$$\begin{array}{l} \langle \Theta \quad\quad\quad\quad\quad ,\, \Gamma,\, S| \circ .x,\, a \quad\quad\quad \rangle \\ \Rightarrow \langle \Theta \oplus [\, a_x \rightarrow x], \Gamma,\, S \quad\quad ,\, a_f([\, a, a_x]) \rangle \\ \textbf{where } a_f = \Phi_a^{\Theta}(\texttt{\_\_getattribute\_\_}) \\ \qquad\quad a_x = \text{new address} \notin \Theta \end{array} \tag{9.11}$$

The `__getattribute__` function is retrieved from the class of the object a. This means that it cannot be overridden by instance attributes.

Getting attributes and instantiating function attributes

The default implementation of `__getattribute__` for objects retrieves an attribute according to the MRO (see Section 5.1). When a class function of an object o is retrieved it is automatically instantiated with the object, i.e., its first argument is set to o. Once instantiated, a class function will not be instantiated again.

We have four cases:

1. If a_x is an instance attribute, we return its value uninstantiated.
2. If a_x is an inherited attribute and is instantiatable, we return the instantiated attribute.
3. If a_x is an inherited attribute but is not instantiatable, we return it.
4. Otherwise the attribute is not defined so we raise an `AttributeError`.

$$
\begin{aligned}
&\langle \Theta,\, \Gamma,\, S,\, \texttt{object.\_\_getattribute\_\_}([\,a_o,\, a_x\,])\rangle \\
\Rightarrow &\begin{cases}
\langle \Theta,\, \Gamma,\, S,\, a & \rangle \text{ if } x \in \Omega^{\Theta}_{a_o} \\
\langle \Theta,\, \Gamma,\, S,\, a_o.a & \rangle \text{ if } x \notin \Omega^{\Theta}_{a_o} \wedge x \in \Phi^{\Theta}_{a_o} \wedge \Theta(a) \text{ is instantiable} \\
\langle \Theta,\, \Gamma,\, S,\, a & \rangle \text{ if } x \notin \Omega^{\Theta}_{a_o} \wedge x \in \Phi^{\Theta}_{a_o} \wedge \Theta(a) \text{ is not instantiable} \\
\langle \Theta,\, \Gamma,\, S,\, \texttt{raise } a_{\texttt{AttributeError}}() & \rangle \text{ otherwise}
\end{cases} \\
&\textbf{where } a = \Upsilon^{\Theta}_{a_o}(x) \\
&\qquad\quad x = \Theta(a_x)
\end{aligned}
\tag{9.12}
$$

Note that only uninstantiated functions are instantiable. Callable objects are instantiated when it is called (see Rule 8.6).

Executing an instantiated function

A function a_f instantiated with object a_o is denoted as $a_o.a_f$. When an instantiated function is evaluated in a function application, the object a_o is inserted as the first argument of the function call.

$$
\begin{aligned}
&\langle \Theta,\, \Gamma,\, S,\, a_o.a_f(\overline{a}) & \rangle \\
\Rightarrow &\langle \Theta,\, \Gamma,\, S,\, a_f(a_o : \overline{a}) & \rangle
\end{aligned}
\tag{9.13}
$$

Getting class members and inserting type checks

The `__getattribute__` primitive of `type` is equivalent to the `__getattribute__` primitive of `object`, except for one difference: we insert type annotations for instantiatable class functions. The type annotations ensure that class functions are only applied to instances of their class.

$$
\begin{aligned}
&\langle \Theta,\, \Gamma,\, S,\, \texttt{type.\_\_getattribute\_\_}([\,a_o,\, a_x\,]) & \rangle \\
\Rightarrow &\begin{cases}
\langle \Theta,\, \Gamma,\, S,\, a : [\,a_o\,] & \rangle \text{ if } x \in \Omega^{\Theta}_{a_o} \wedge \Theta(a) \text{ is instantiable} \\
\langle \Theta,\, \Gamma,\, S,\, \texttt{object.\_\_getattribute\_\_}([\,a_o,\, a_x\,]) & \rangle \text{ otherwise}
\end{cases} \\
&\textbf{where } a = \Upsilon^{\Theta}_{a_o}(x) \\
&\qquad\quad x = \Theta(a_x)
\end{aligned}
\tag{9.14}
$$

9.4 Attribute assignment

By default, the attribute assignment statement $e_o.x = e$ (re)defines the attribute x of the target object e_o to the value e. The semantics of attribute assignment statements are determined by the `__setattr__` attribute of an object's class. By overloading it, a programmer can redefine the semantics.

Evaluating the right hand side of an attribute assignment

The evaluation of the assignment starts with the right hand side: the value e. Note that this behaviour is different from the usual left to right evaluation.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S \qquad\qquad ,\, e_o.x = e; \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S | e_o.x = \circ,\, e \qquad\quad \rangle \end{array} \tag{9.15}$$

Evaluating the left hand side of an attribute assignment

Next, the left hand side e_o is evaluated, while the value a of the right hand side is placed on the stack.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S | e_o.x = \circ,\, a \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S | \circ .x = a,\, e_o \rangle \end{array} \tag{9.16}$$

Calling `__setattr__`

Finally we call the inherited attribute `__setattr__` of a_o, with the target object a_o, the assigned attribute name a_x, and the new value a.

$$\begin{array}{l} \langle \Theta \qquad\qquad ,\, \Gamma,\, S | \circ .x = a,\, a_o \qquad\quad \rangle \\ \Rightarrow \langle \Theta \oplus [a_x \to x],\, \Gamma,\, S \qquad\quad ,\, a_f([a_o, a_x, a]) \rangle \\ \textbf{where } a_f = \Phi^{\Theta}_{a_o}(\texttt{\_\_setattr\_\_}) \\ \qquad\quad a_x = \text{new address} \notin \Theta \end{array} \tag{9.17}$$

The default implementation of `__setattr__` for objects

The default implementation of `__setattr__` for objects updates the object a_o with $[x \to a]$.

$$\begin{array}{l} \langle \Theta \qquad\qquad ,\, \Gamma,\, S,\, \texttt{object.\_\_setattr\_\_}([a_o, a_x, a]) \rangle \\ \Rightarrow \langle \Theta \oplus [a_o \to o'],\, \Gamma,\, S,\, a_{\texttt{None}} \qquad\qquad \rangle \\ \textbf{where } o' = \Theta(a_o) \oplus [\Theta(a_x) \to a] \end{array} \tag{9.18}$$

9.5 Attribute deletion

By default, an attribute deletion statement `del` $e_o.x$ removes the attribute x from object e_o if it exists. The semantics can be redefined by overloading `__delattr__`.

Evaluating the left hand side of an attribute deletion

We start by evaluating the object e_o.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S \qquad\qquad ,\, \texttt{del}\ e_o.x; \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|\texttt{del}\ \circ.x,\, e_o \qquad\quad \rangle \end{array} \tag{9.19}$$

Calling `__delattr__`

When e_o has been evaluated to a_o, we call the `__delattr__` attribute inherited by a_o.

$$\begin{array}{l} \langle \Theta \qquad\qquad ,\, \Gamma,\, S|\texttt{del}\ \circ.x,\, a_o \qquad\quad \rangle \\ \Rightarrow \langle \Theta \oplus [\, a_x \to x\,],\, \Gamma,\, S \qquad\quad ,\, a_f([\, a_o, a_x\,]) \rangle \\ \textbf{where}\ a_f = \Phi^{\Theta}_{a_o}(\texttt{\_\_delattr\_\_}) \\ \qquad\ a_x = \text{new address} \notin \Theta \end{array} \tag{9.20}$$

The default implementation of `__delattr__`

The default implementation of `__delattr__` removes the attribute a_x from the object a_o if x is defined in a_o. Otherwise an `AttributeError` is raised.

$$\begin{array}{l} \quad \langle \Theta \qquad\qquad ,\, \Gamma,\, S,\, \texttt{object.\_\_delattr\_\_}([\, a_o, a_x\,]) \rangle \\ \Rightarrow \begin{cases} \langle \Theta \oplus [\, a_o \to o'\,],\, \Gamma,\, S,\, a_{\texttt{None}} \qquad\quad \rangle & \text{if } \Theta(a_x) \in \Theta(a_o) \\ \langle \Theta \qquad\qquad ,\, \Gamma,\, S,\, \texttt{raise}\ a_{\texttt{AttributeError}}() \qquad \rangle & \text{otherwise} \end{cases} \\ \textbf{where}\ o' = \Theta(a_o) \ominus \Theta(a_x) \end{array} \tag{9.21}$$

10 Exceptions

Exceptions simplify handling of errors by providing an alternative mode of execution. When an exceptional situation occurs i.e., an exception is raised, normal execution stops, until the situation is dealt with.

Listing 10.1 shows how exceptions can be used to handle divisions by zero, how normal execution stops when an exception is raised, and how the exception is handled.

```
class ZeroDivisionError(Exception) :
    def __str__(self) :
        return "Divided␣a␣number␣by␣zero!"

def div(x, y) :
    if y == 0 :
        raise ZeroDivisionError()

    return x / y

try :
    div(2, 0) # will raise the exception
    print("This␣will␣never␣be␣printed")
except Exception, e : # catch all exceptions from within the try block
    print(e)

print("Normal␣exectution␣continues")
```

Listing 10.1: Raising and catching a divide by zero exception

10.1 The raise statement

A raise statement `raise` e stops normal execution and raises e if and only if it evaluates to an instance of `BaseException`. During normal execution we would continue with the next frame on the stack, whereas most stack frames are simply discarded while raising an exception. Only some stack frames and markers will be handled while raising an exception.

Executing the raise statement

The raise statement first evaluates the exception expression e.

$$\begin{aligned} &\langle \Theta, \Gamma, S \quad\quad\quad\quad\quad\quad , \texttt{raise}\ e; \rangle \\ \Rightarrow\ &\langle \Theta, \Gamma, S|\texttt{raise}\ \circ , e \quad\quad\quad\quad \rangle \end{aligned} \tag{10.1}$$

Start unwinding of the stack

If the raised value a is an instance of `BaseException`, unwinding of the stack starts. Otherwise, a `TypeError` is raised. Raising an exception has no effect on the environment.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S | \texttt{raise}\ \circ,\, a \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S,\, \texttt{raise}\ a \rangle & \text{if } (a :_\Theta a_{\texttt{BaseException}}) \\ \langle \Theta,\, \Gamma,\, S,\, \texttt{raise}\ a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \end{array} \tag{10.2}$$

Raising an exception

By default, stack frames are simply discarded while unwinding the stack until the stack is empty and execution stops. Note that this is a very unspecific rule and hence will be overridden by rules such as Rule 10.4.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S | f,\, \texttt{raise}\ a \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S,\, \texttt{raise}\ a \rangle \end{array} \tag{10.3}$$

Raising an exception in a function

If an exception remains uncaught in a function, we return from the function by reinstating the environment Γ' found on the return marker introduced in Rule 8.6, and resume unwinding.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S | \Gamma' \vdash \circ_\lambda,\, \texttt{raise}\ a \rangle \\ \Rightarrow \langle \Theta,\, \Gamma',\, S,\, \texttt{raise}\ a \rangle \end{array} \tag{10.4}$$

Raising an exception in a generator

When an exception is raised in a generator, we will encounter the yield marker $\Gamma' \vdash a_g$ introduced in Rule 8.11. Once a generator has raised an exception, it is stopped: the current generator a_g is updated with an empty generator $\langle \Box, \Box \rangle$ and the old environment Γ' is reinstated (see also Rule 8.16).

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S | \Gamma' \vdash a_g,\, \texttt{raise}\ a \rangle \\ \Rightarrow \langle \Theta \oplus [\, a_g \to \langle \Box, \Box \rangle\,],\, \Gamma',\, S,\, \texttt{raise}\ a \rangle \end{array} \tag{10.5}$$

10.2 The try-except statement

A raised exception is handled by a try-except statement. A try-except statement `try` : b `except` e`,` x : b_e executes its body b. If an exception is raised in the body and it is an instance of e, the raised exception is bound to x and normal execution resumes with the except clause b_e. If a raised exception is not an instance of e, the exception is re-raised and unwinding continues.

Executing the try-except statement

The try-except statement places the except clause `except` e`,` x : b_e on the stack, and executes the body b.

$$\begin{array}{l} \langle \Theta, \Gamma, S \qquad\qquad\qquad\quad , \texttt{try} : b\ \texttt{except}\ e\texttt{,}\ x : b_e; \rangle \\ \Rightarrow \langle \Theta, \Gamma, S|\texttt{except}\ e\texttt{,}\ x : b_e, b \qquad\qquad\quad \rangle \end{array} \tag{10.6}$$

Popping an exception clause while unwinding

When an except clause is encountered while unwinding, its exception class e is evaluated in order to check whether it catches the exception. The raised exception a is put on the stack, along with the exception clause.

$$\begin{array}{l} \langle \Theta, \Gamma, S|\texttt{except}\ e\texttt{,}\ x : b_e, \texttt{raise}\ a \rangle \\ \Rightarrow \langle \Theta, \Gamma, S|{}^{\nearrow \texttt{raise}\ a}_{\texttt{except}\ \circ\texttt{,}\ x:b_e} \quad , e \qquad \rangle \end{array} \tag{10.7}$$

Catching an exception

Once the class of the except clause has been evaluated, we determine whether it catches the raised exception:

1. If the class a_e of the except clause is an exception (i.e. it is a `BaseException`) and the raise exception a is an instance of a_e, we catch the exception. The exception a is bound to x and the b_e is executed.
2. Otherwise, we re-raise the exception a and pop the except clause.

$$\begin{array}{l} \quad \langle \Theta\ , \Gamma|\gamma_1, S|{}^{\nearrow \texttt{raise}\ a}_{\texttt{except}\ \circ\texttt{,}\ x:b_e}, a_e \qquad \rangle \\ \Rightarrow \begin{cases} \langle \Theta', \Gamma|\gamma_1, S \qquad\qquad\quad , b_e \qquad \rangle & \text{if } a_e <:_\Theta a_{\texttt{BaseException}} \wedge a :_\Theta a_e \\ \langle \Theta\ , \Gamma|\gamma_1, S \qquad\qquad\quad , \texttt{raise}\ a \rangle & \text{otherwise} \end{cases} \\ \textbf{where } \Theta' = \Theta \oplus [\gamma_1 \to \Theta(\gamma_1) \oplus [x \to a]] \end{array} \tag{10.8}$$

The class of an except clause a_e can be any value. It is not at all limited to classes, or even exceptions. However, if a_e is not an exception class, it will never catch anything.

Popping the exception marker

The exception clause is simply popped from the stack when no exception has been raised in the body.

$$\begin{aligned} &\langle \Theta, \Gamma, S | \texttt{except}\ e\texttt{,}\ x : b_e,\ a \rangle \\ \Rightarrow\ &\langle \Theta, \Gamma, S\ ,\ a \rangle \end{aligned} \tag{10.9}$$

10.3 The try-finally statement

Sometimes it is necessary to perform some operations even when an error has occurred. The try-finally statement $\texttt{try} : b\ \texttt{finally} : b_f$ executes its body b with a guarantee that b_f will be executed. We would typically use a try-finally statement to ensure some clean-up code is executed, even if an exception is raised in its body.

The guarantee of executing b_f applies not only to exception handling. For example, if there is a return statement in the body b, we will execute b_f before we return from the function. If b_f contains another return statement, the last return statement will return from the function. This rather unintuitive behaviour has prompted the CPython developers to give a warning when the b_f contains a yield or return statement.

Executing the try-finally statement

The try-finally starts like the try-except statement, by executing its body. The finally block b_f is placed on the stack.

$$\begin{aligned} &\langle \Theta, \Gamma, S\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ , \texttt{try} : b\ \texttt{finally} : b_f; \rangle \\ \Rightarrow\ &\langle \Theta, \Gamma, S | \texttt{finally} : b_f,\ b\ \rangle \end{aligned} \tag{10.10}$$

Popping a finally clause while unwinding

When a finally clause in encountered while unwinding, the raised exception a is put on the stack and the finally clause b_f is executed.

$$\begin{aligned} &\langle \Theta, \Gamma, S | \texttt{finally} : b_f, \texttt{raise}\ a_e \rangle \\ \Rightarrow\ &\langle \Theta, \Gamma, S | \texttt{raise}\ a_e\ \ \ \ , b_f\ \ \ \ \ \ \ \ \rangle \end{aligned} \tag{10.11}$$

Popping the finally marker

When no exception has occurred, the try-finally statement executes the finally clause that has been placed on the stack.

$$\begin{array}{l} \left\langle \Theta,\, \Gamma,\, S|\texttt{finally} : b_f,\, a \right\rangle \\ \Rightarrow \left\langle \Theta,\, \Gamma,\, S \qquad\qquad\quad ,\, b_f \right\rangle \end{array} \tag{10.12}$$

11 Control structures

Python includes three basic control structures: if, while and for statements. All three are very common in imperative programming languages and have similar semantics in Python. The if statement executes one of its two blocks of statements, depending on the value of a test expression. The while expression repeats its body as long as its test expression is nonzero. The for statement iterates over a number of values, executing its body once for every value.

Unlike many other languages, the control structures of Python are non-scoping compound statements, i.e., statements that contain blocks of statements but do not define a lexical scope (see Chapter 3.2). This means that the variables assigned inside an if, while or for statement are still bound after the execution of the statements.

The break and continue statements, respectively stop or interrupt a loop. They can only occur in the body of a for or while loop.

11.1 The if statement

An if statement $\texttt{if}\ e : b_{\textbf{if}}\ \texttt{else} : b_{\textbf{else}}$ consists of the test expression e, the blocks $b_{\textbf{if}}$ and $b_{\textbf{else}}$. If e evaluates to a nonzero value, $b_{\textbf{if}}$ is executed, else $b_{\textbf{else}}$ is executed. We consider a value to be nonzero, if its class function `__nonzero__` returns `True`.

Evaluating the test expression of an if statement

First, the test expression e is evaluated, while the frame $\texttt{if}\ \circ.\texttt{\_\_nonzero\_\_}() : b_{\textbf{if}}\ \texttt{else} : b_{\textbf{else}}$ indicates that we still need to call `__nonzero__` and execute either $b_{\textbf{if}}$ or $b_{\textbf{else}}$.

$$
\begin{aligned}
&\big\langle \Theta, \Gamma, S \qquad\qquad\qquad\qquad\qquad\qquad , \texttt{if}\ e : b_{\textbf{if}}\ \texttt{else} : b_{\textbf{else}} ; \big\rangle \\
\Rightarrow &\big\langle \Theta, \Gamma, S | \texttt{if}\ \circ.\texttt{\_\_nonzero\_\_}() : b_{\textbf{if}}\ \texttt{else} : b_{\textbf{else}}, e \qquad\qquad\qquad\qquad \big\rangle
\end{aligned}
\tag{11.1}
$$

Testing the test value of an if statement

Next, we call the class function `__nonzero__` to determine which block should be executed. The stack frame $\texttt{if}\ \circ : b_{\textbf{if}}\ \texttt{else} : b_{\textbf{else}}$ indicates that we are evaluating $a.\texttt{\_\_nonzero\_\_}\ ()$ and contains the two blocks. If the test value a has no class attribute `__nonzero__`, we raise a `TypeError`.

$$
\begin{array}{l}
\quad \langle \Theta,\, \Gamma,\, S | \texttt{if}\ \circ\ \texttt{.\_\_nonzero\_\_}() : b_{\,\textbf{if}}\ \texttt{else} : b_{\,\textbf{else}}\,,\, a \rangle \\
\Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S | \texttt{if}\ \circ : b_{\,\textbf{if}}\ \texttt{else} : b_{\,\textbf{else}} \,,\, a_f([a]) \rangle & \text{if } \texttt{\_\_nonzero\_\_} \in \Phi_a^\Theta \\ \langle \Theta,\, \Gamma,\, S \,,\, \texttt{raise}\ a_{\texttt{TE}}() \rangle & \text{otherwise} \end{cases} \\
\textbf{where } a_f = \Phi_a^\Theta(\texttt{\_\_nonzero\_\_}) \\
\qquad\quad a_{\texttt{TE}} = a_{\texttt{TypeError}}
\end{array}
\tag{11.2}
$$

Executing the body of an if statement

Depending on the value of the test expression we execute either the **if** or the **else** block:

1. If `__nonzero__` returned a boolean, and it is `True`, we execute $b_{\,\textbf{if}}$.
2. If `__nonzero__` returned a boolean, and it is `False`, we execute $b_{\,\textbf{else}}$.
3. Otherwise, the call to `__nonzero__` returned a different class, so we return a `TypeError`.

$$
\begin{array}{l}
\quad \langle \Theta,\, \Gamma,\, S | \texttt{if}\ \circ : b_{\,\textbf{if}}\ \texttt{else} : b_{\,\textbf{else}}\,,\, a \rangle \\
\Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S \,,\, b_{\,\textbf{if}} \rangle & \text{if } a :_\Theta a_{\texttt{bool}} \wedge \Theta(a) \\ \langle \Theta,\, \Gamma,\, S \,,\, b_{\,\textbf{else}} \rangle & \text{if } a :_\Theta a_{\texttt{bool}} \wedge \neg\, \Theta(a) \\ \langle \Theta,\, \Gamma,\, S \,,\, \texttt{raise}\ a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases}
\end{array}
\tag{11.3}
$$

11.2 The while statement

The while statement `while` $e : b$ consists of a test expression e and a body b. Like the if statement, the while statement depends on the value of the test expression: if it evaluates to a nonzero value, the body is executed and the while statement (including the test) is executed again, otherwise the while-statement ends.

Evaluating the test expression of a while statement

First, we evaluate the test expression e and place the while statement on the stack. The frame `while` $\circ_e$ `.__nonzero__()` : b indicates that we still have to call `__nonzero__` and contains both the body and test expression.

$$
\begin{array}{l}
\quad \langle \Theta,\, \Gamma,\, S \,,\, \texttt{while}\ e : b; \rangle \\
\Rightarrow \langle \Theta,\, \Gamma,\, S | \texttt{while}\ \circ_e \texttt{.\_\_nonzero\_\_}() : b,\, e \rangle
\end{array}
\tag{11.4}
$$

Testing the test value of a while statement

Next, we call the class function `__nonzero__` of the test value a, or, if it has no such attribute, raise a `TypeError`.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|\texttt{while}\ \circ_e .\texttt{\_\_nonzero\_\_}() : b,\, a \qquad\qquad\quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S|\texttt{while}\ \circ_e : b \qquad\quad ,\, a_f([a]) \qquad\quad \rangle & \text{if } \texttt{\_\_nonzero\_\_} \in \Phi_a^\Theta \\ \langle \Theta,\, \Gamma,\, S \qquad\qquad\qquad\quad ,\, \texttt{raise}\ a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \\ \textbf{where}\ a_f = \Phi_a^\Theta(\texttt{\_\_nonzero\_\_}) \end{array} \tag{11.5}$$

Executing the body of a while statement

We execute the body b and rerun the while loop, if the test expression evaluated to a nonzero value. If not, we stop the loop:

1. If `__nonzero__` returned a boolean, and it is `True`, we execute the body, and place a while marker `while` $e : \circ_b$ on the stack.
2. If `__nonzero__` returned a boolean, and it is `False`, we stop of the while statement and return `None`.
3. Otherwise, the call to `__nonzero__` returned a different class, so we return a `TypeError`.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|\texttt{while}\ \circ_e : b,\, a \qquad\qquad\quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S|\texttt{while}\ e : \circ_b,\, b \qquad\quad \rangle & \text{if } a :\Theta a_{\texttt{bool}} \wedge \Theta(a) \\ \langle \Theta,\, \Gamma,\, S \qquad\qquad\quad ,\, a_{\texttt{None}} \qquad\quad \rangle & \text{if } a :\Theta a_{\texttt{bool}} \wedge \neg\, \Theta(a) \\ \langle \Theta,\, \Gamma,\, S \qquad\qquad\quad ,\, \texttt{raise}\ a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \end{array} \tag{11.6}$$

Repeating a while statement

After the execution of the while body, the while marker on top of the stack is executed again.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|\texttt{while}\ e : \circ_b,\, a \qquad\quad \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S \qquad\qquad\quad ,\, \texttt{while}\ e : b; \rangle \end{array} \tag{11.7}$$

One might think that the while marker can be replaced by a simple while statement on the stack. However, the rules that handle continue and break statements depend on the difference (see Sections 11.4 and 11.5).

11.3 The for statement

A for statement `for` x `in` $e : b$ consists of a variable name x to which the values produced by the iterable e are bound, and the body b.

An iterable is any object with a class function `__iter__` that produces an iterator: an object with a class function `__next__` that returns the next value if there is one, and raises a `StopIteration` exception when not. In particular, every generator is an iterator and every generator function is an iterable.

A for statement first creates an iterator by calling the class function `__iter__` of the iterable e. Then it executes the body b repeatedly, calling the `__next__` function of the iterator. Binding the variable x to the value returned by `__next__`. Once the `__next__` function raises a `StopIteration` exception, the for statement stops.

Executing the for statement

First, the iterable e is evaluated and the rest of the for statement is put on the stack. The stack frame `for` x `in` $\circ$`.__iter__()` $: b$ indicates that we need to call e`.__iter__`.

$$\begin{aligned} &\langle \Theta,\, \Gamma,\, S &&,\, \texttt{for}\ x\ \texttt{in}\ e : b; \rangle \\ \Rightarrow\ &\langle \Theta,\, \Gamma,\, S|\texttt{for}\ x\ \texttt{in}\ \circ\texttt{.\_\_iter\_\_}() : b,\, e &&\rangle \end{aligned} \tag{11.8}$$

Get the iterator of the iterable

Next, if the iterable has an `__iter__` attribute, we call it, otherwise we raise a `TypeError`.

$$\begin{aligned} &\langle \Theta,\, \Gamma,\, S|\texttt{for}\ x\ \texttt{in}\ \circ\texttt{.\_\_iter\_\_}() : b,\, a &&\rangle \\ \Rightarrow\ &\begin{cases} \langle \Theta,\, \Gamma,\, S|\texttt{for}\ x\ \texttt{in}\ \circ : b &,\, a_f([\,a\,]) &\rangle & \text{if } \texttt{\_\_iter\_\_} \in \Phi_a^\Theta \\ \langle \Theta,\, \Gamma,\, S &,\, \texttt{raise}\ a_{\texttt{TypeError}}() &\rangle & \text{otherwise} \end{cases} \\ &\textbf{where } a_f = \Phi_a^\Theta(\texttt{\_\_iter\_\_}) \end{aligned} \tag{11.9}$$

Start the for loop with the iterator

Then, we start the for loop with the iterator a_i, that has been returned by `__iter__`, and push a for marker `for` x `in` $a_i : \circ_b$ on the stack to start an iteration.

$$\begin{aligned} &\langle \Theta,\, \Gamma,\, S|\texttt{for}\ x\ \texttt{in}\ \circ : b \ \ ,\, a_i \ \ \rangle \\ \Rightarrow\ &\langle \Theta,\, \Gamma,\, S|\texttt{for}\ x\ \texttt{in}\ a_i : \circ_b,\, a_{\texttt{None}} \rangle \end{aligned} \tag{11.10}$$

The notation of the for marker suggests that the body is being executed. This is clearly not the case, instead, we pretend to just have executed the body in order to trigger Rule 11.11.

Starting an iteration of the for loop

An iteration starts with a call to the `__next__` function attribute of the iterator a_i. If no such attribute is available, then a_i is no iterator, so a `TypeError` is raised.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S | \texttt{for}\ x\ \texttt{in}\ a_i : \circ_b,\, a \qquad\qquad\quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S | \texttt{for}\ x\ \texttt{in}\ a_i : b\ ,\, a_f([\,a_i\,]) \qquad\quad \rangle & \text{if } \texttt{\_\_next\_\_} \in \Phi^{\Theta}_{a_i} \\ \langle \Theta,\, \Gamma,\, S \qquad\qquad\qquad\quad ,\, \texttt{raise}\ a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a_f = \Phi^{\Theta}_{a_i}(\texttt{\_\_next\_\_}) \end{array} \tag{11.11}$$

Executing the body of a for loop

The value a returned by the iterator, is bound to x and the body b is executed. We schedule the next iteration by pushing a for marker `for` x `in` $a_i : \circ_b$ on the stack.

$$\begin{array}{l} \langle \Theta \qquad\qquad\quad ,\, \Gamma | \gamma_1,\, S | \texttt{for}\ x\ \texttt{in}\ a_i : b\ ,\, a \rangle \\ \Rightarrow \langle \Theta \oplus [\gamma_1 \to m],\, \Gamma | \gamma_1,\, S | \texttt{for}\ x\ \texttt{in}\ a_i : \circ_b,\, b \rangle \\ \textbf{where } m = \Theta(\gamma_1) \oplus [x \to a] \end{array} \tag{11.12}$$

Stopping the for loop

The for loop stops, when the iterator's `__next__` function attribute raises a `StopIteration`: If we encounter a stack frame `for` x `in` $a_i : b$ while unwinding the stack with an exception a, we check whether the exception a is a `StopIteration`. If it is, normal execution resumes and we return `None`. Otherwise, the frame is popped and unwinding continues.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S | \texttt{for}\ x\ \texttt{in}\ a_i : b, \texttt{raise}\ a \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S \qquad\qquad\quad ,\, a_{\texttt{None}} \rangle & \text{if } a :\Theta a_{\texttt{StopIteration}} \\ \langle \Theta,\, \Gamma,\, S \qquad\qquad\quad ,\, \texttt{raise}\ a \rangle & \text{otherwise} \end{cases} \end{array} \tag{11.13}$$

11.4 The break statement

A break statement immediately terminates execution of the current block and continues execution of the block that encapsulates the while or for statement. In other words, a break statement jumps directly to the end of the loop.

Rewinding the stack for a break statement

Break statements unwind the stack until a frame indicating the end of a while or for loop is encountered. If we encounter a finally clause `finally` : b (see Section 10.10), we execute its

body b and put the break statement on the stack. When b has been executed, unwinding will continue with the execution of the break statement.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S|f \quad\quad\quad\quad ,\, \texttt{break;} \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S \quad\quad\quad\quad ,\, a_{\texttt{None}} \rangle & \text{if } f = \texttt{while } e : \circ_b \\ \langle \Theta,\, \Gamma,\, S \quad\quad\quad\quad ,\, a_{\texttt{None}} \rangle & \text{if } f = \texttt{for } x \texttt{ in } a_i : \circ_b \\ \langle \Theta,\, \Gamma,\, S|\texttt{break;} ,\, b \quad\quad\quad\quad \rangle & \text{if } f = \texttt{finally} : b \\ \langle \Theta,\, \Gamma,\, S \quad\quad\quad\quad ,\, \texttt{break;} \rangle & \text{otherwise} \end{cases} \end{array} \tag{11.14}$$

11.5 The continue statement

Like a break statement, a continue statement terminates execution of a loop's body. But rather than jumping to the end of the loop, it starts another iteration.

Rewinding the stack for a continue statement

Continue statements unwind the stack until the end of a for or while loop. At the end of a loop, the loop is executed again. Like with break statements, any finally clauses will be executed 'on the way out'.

In case of a for or while loop, the for or while marker is left on the stack, and normal execution is resumed. This will cause Rule 11.7 or Rule 11.11 respectively, to run another iteration.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S|f \quad\quad\quad\quad\quad\quad ,\, \texttt{continue;} \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S|\texttt{while } e : \circ_b \quad ,\, a_{\texttt{None}} \quad\quad \rangle & \text{if } f = \texttt{while } e : \circ_b \\ \langle \Theta,\, \Gamma,\, S|\texttt{for } x \texttt{ in } a_i : \circ_b ,\, a_{\texttt{None}} \quad\quad \rangle & \text{if } f = \texttt{for } x \texttt{ in } a_i : \circ_b \\ \langle \Theta,\, \Gamma,\, S|\texttt{continue;} \quad ,\, b \quad\quad\quad \rangle & \text{if } f = \texttt{finally} : b \\ \langle \Theta,\, \Gamma,\, S \quad\quad\quad\quad\quad\quad ,\, \texttt{continue;} \rangle & \text{otherwise} \end{cases} \end{array} \tag{11.15}$$

12 Print statements

The print statement `print`(e) prints a string representation of e to the standard output. The string representation of e is determined by its class function `__str__`. The program in Listing 12.1 defines an object with a special string representation.

```
class Person(object) :
       def __init__(self, name, age) :
              self.name = name
              self.age = age
       def __str__(self) :
              return self.name + "␣is␣" + self.age.__str__() + "␣years␣old"

print(Person("Gideon", 25)) # prints 'Gideon is 25 years old'
```

Listing 12.1: A simple Person class with a custom string representation

Classes that do not define their own string representation inherit the `__str__` attribute from their superclass(es). Every object has a string representation because all classes inherit `__str__` from object. The primitive `object.__str__` (specified in Rule A.1) uses an objects address to create a string representation that is unique for every object.

Evaluating the printed expression

First, the expression e to be printed is evaluated.

$$\begin{array}{rl} & \langle \Theta,\, \Gamma,\, S & ,\, \texttt{print}(e); \rangle \\ \Rightarrow & \langle \Theta,\, \Gamma,\, S|\texttt{print}(\circ.\texttt{\_\_str\_\_}()),\, e & \rangle \end{array} \tag{12.1}$$

Getting a string representation of the printed value

Next, we call the class function `__str__` of a to obtain a string representation of a. We don't need to check whether an object has a class function `__str__`, unlike for example the `__iter__` attribute in Rule 11.9.

$$\begin{array}{rl} & \langle \Theta,\, \Gamma,\, S|\texttt{print}(\circ.\texttt{\_\_str\_\_}()),\, a & \rangle \\ \Rightarrow & \langle \Theta,\, \Gamma,\, S|\texttt{print}(\circ) & ,\, a_f([a]) \rangle \\ \textbf{where} & a_f = \Phi_a^{\Theta}(\texttt{\_\_str\_\_}) & \end{array} \tag{12.2}$$

Printing the string

Finally, the string representation $\Theta(a)$ of the object can be printed. Provided that the class function `__str__` returned a string.

$$\xrightarrow{\text{print } \Theta(a)} \begin{array}{l} \langle \Theta,\, \Gamma,\, S|\texttt{print}(\circ),\, a \rangle \text{ if } a :_\Theta a_{\texttt{str}} \\ \langle \Theta,\, \Gamma,\, S,\, a_{\texttt{None}} \rangle \end{array} \tag{12.3}$$

Faulty string representation

If the class function `__str__` did not return a string, we raise a type error.

$$\Rightarrow \begin{array}{l} \langle \Theta,\, \Gamma,\, S|\texttt{print}(\circ),\, a \rangle \text{ if } \neg\, (a :_\Theta a_{\texttt{str}}) \\ \langle \Theta,\, \Gamma,\, S,\, \texttt{raise } a_{\texttt{TypeError}}() \rangle \end{array} \tag{12.4}$$

13 Operators

The semantics of most operators are determined by the class function of their operands. For example the addition operator + is implemented by the class function `__add__` of the left operand. An expression $e_l + e_r$ could[1] be translated to $e_l.\texttt{\_\_add\_\_}([e_r])$.

Boolean operators are an exception to this rule. They are not implemented by class function but depend on the class function `__nonzero__` to coerce values to booleans.

13.1 Unary operators

Unary operators have only one operand. Evaluation takes three simple steps: evaluation of the operand, getting the operator's attribute from the operand's class, execute the class function.

Evaluating the operand of a unary operator expression

$$\begin{array}{l} \langle \Theta, \Gamma, S \quad\quad\quad , \bigcirc e \rangle \\ \Rightarrow \langle \Theta, \Gamma, S|\bigcirc \circ, e \quad\quad \rangle \end{array} \tag{13.1}$$

Evaluating the unary not operator

The `not` operator negates the value returned by the class function `__nonzero__` of its operand. If the operand has no attribute `__nonzero__`, we raise a `TypeError`.

$$\begin{array}{l} \quad \langle \Theta, \Gamma, S|\texttt{not} \circ \quad\quad\quad\quad , a \quad\quad\quad\quad\quad\quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta, \Gamma, S|\texttt{not} \circ_{\texttt{\_\_nonzero\_\_}} , a_f([a]) \quad\quad\quad\quad\quad \rangle & \text{if } \texttt{\_\_nonzero\_\_} \in \Phi_a^\Theta \\ \langle \Theta, \Gamma, S \quad\quad\quad\quad\quad\quad\quad , \texttt{raise } a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a_f = \Phi_a^\Theta(\texttt{\_\_nonzero\_\_}) \end{array} \tag{13.2}$$

[1] This simple rewrite of the expression has a different order of evaluation and assumes `__getattribute__` is not overridden.

Return the negated value of a not operand

If __nonzero__ returned a boolean, we return its negated value. Otherwise, if __nonzero__ returned something else, we raise a `TypeError`.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|\texttt{not} \circ_{\texttt{\_\_nonzero\_\_}},\, a \qquad\qquad\quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S \qquad\qquad\quad ,\, \neg\, \Theta(a) \qquad\quad \rangle & \text{if } a :\Theta a_{\texttt{bool}} \\ \langle \Theta,\, \Gamma,\, S \qquad\qquad\quad ,\, \texttt{raise } a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \end{array} \tag{13.3}$$

Executing the function of a unary operator

For other operators, we call the appropriate operator function.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|\bigcirc \circ,\, a \qquad\qquad\quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S \qquad ,\, a_f([\,a\,]) \qquad\quad \rangle & \text{if attribute of } \bigcirc \in \Phi_a^\Theta \\ \langle \Theta,\, \Gamma,\, S \qquad ,\, \texttt{raise } a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a_f = \Phi_a^\Theta(\text{attribute of } \bigcirc) \end{array} \tag{13.4}$$

13.2 Binary operators

Most operators have two operands: a left and a right hand side operator. The semantics of the operator are defined by the left hand side operator.

Evaluating the left hand side of a binary operator

First, the left hand side operator is evaluated while the remaining expression is put on the stack.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, e_l \bigcirc e_r \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|\circ \bigcirc e_r,\, e_l \qquad \rangle \end{array} \tag{13.5}$$

Evaluating the right hand side of a binary operator

The boolean operators `and` and `or` are lazy. They only evaluate their right hand side if needed. Therefore, we call __nonzero__ for the `and` and `or` operators before e_r is evaluated. If __nonzero__ is not defined, we raise a `TypeError`.

Other operators are executed after both their operators have been evaluated. Therefore, we evaluate e_r and store a_l on the stack.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|\circ \bigcirc e_r \;\;,\, a_l \qquad\qquad\;\; \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S|\circ \bigcirc_B e_r,\, a_f([a_l]) \qquad \rangle & \text{if } \bigcirc \in [\texttt{and}, \texttt{or}] \wedge \texttt{\_\_nonzero\_\_} \in \Phi^{\Theta}_{a_l} \\ \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, \texttt{raise } a_{\texttt{TypeError}}() \rangle & \text{if } \bigcirc \in [\texttt{and}, \texttt{or}] \wedge \texttt{\_\_nonzero\_\_} \notin \Phi^{\Theta}_{a_l} \\ \langle \Theta,\, \Gamma,\, S|a_l \bigcirc \circ \;\;,\, e_r \qquad\quad \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a_f = \Phi^{\Theta}_{a_l}(\texttt{\_\_nonzero\_\_}) \end{array} \tag{13.6}$$

Evaluation of the boolean operators continues at Rules 13.9 and 13.10. The `in` operator continues at Rule 13.7. Other operators are evaluated by Rule 13.8.

Evaluating an *is* expression

The `is` operator determines whether its operands are the same object. This is achieved by comparing the addresses of the operands.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S|a_l \texttt{is} \circ\,,\, a_r \qquad \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, a_l \equiv a_r \rangle \end{array} \tag{13.7}$$

Calling the operator attribute

When both operands of the operator have been evaluated, the operator's class function of a_l is called with the operands as its arguments.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|a_l \bigcirc \circ,\, a_r \qquad\qquad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, a_f([a_l, a_r]) \qquad \rangle & \text{if attribute of } \bigcirc \in \Phi^{\Theta}_{a_l} \\ \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, \texttt{raise } a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a_f = \Phi^{\Theta}_{a_l}(\text{attribute of } \bigcirc) \end{array} \tag{13.8}$$

Lazy execution of the `and` operator

The `and` expression returns e_r if the e_l is not zero (its `__nonzero__` returned `True`), otherwise it returns `False`. Note that the right hand side e_r can evaluate to any value.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|\circ \texttt{and}_B e_r,\, a \qquad\qquad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, e_r \qquad\qquad \rangle & \text{if } a :\Theta a_{\texttt{bool}} \wedge \Theta(a) \\ \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, \texttt{False} \qquad\quad \rangle & \text{if } a :\Theta a_{\texttt{bool}} \wedge \neg\, \Theta(a) \\ \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, \texttt{raise } a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \end{array} \tag{13.9}$$

Lazy execution of the `or` operator

Conversely, the `or` expression returns e_r when e_l is zero and True otherwise.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|\circ \texttt{or}_B\, e_r,\, a \qquad\qquad\quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S \quad\quad ,\, \texttt{True} \qquad\quad \rangle & \text{if } a :\Theta\, a_{\texttt{bool}} \wedge \Theta(a) \\ \langle \Theta,\, \Gamma,\, S \quad\quad ,\, e_r \qquad\quad \rangle & \text{if } a :\Theta\, a_{\texttt{bool}} \wedge \neg\, \Theta(a) \\ \langle \Theta,\, \Gamma,\, S \quad\quad ,\, \texttt{raise } a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \end{array} \tag{13.10}$$

13.3 Container operators

Containers are values containing values, or actually addresses pointing to values. Individual items of a container can be retrieved, set, or deleted using the container operators. The semantics of those operators depend on the implementation of the container's class functions `__setitem__`, `__getitem__`, and `__delitem__`.

For example, lists, tuples and dictionaries are containers with their own implementation of the container attributes (see Appendix A). Not just built-in values are containers though, any object with the right class functions is considered a container.

Retrieving a container item

The subscription $e[e_i]$ retrieves an item with key or index e_i from a container e.

Evaluating the container expression

First, the container e is evaluated.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S \quad\quad ,\, e[e_i] \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|\circ [\, e_i],\, e \quad\; \rangle \end{array} \tag{13.11}$$

Evaluating the key expression

Then, the index or key e_i is evaluated.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|\circ [\, e_i],\, a \,\rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|a[\circ] \;\; ,\, e_i \rangle \end{array} \tag{13.12}$$

Calling __getitem__ of a container

Finally, the container's class function __getitem__ is called to retrieve the item.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|a[\circ],\, a_i \qquad\qquad\quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S \quad ,\, a_f([a, a_i]) \qquad \rangle & \text{if } \texttt{\_\_getitem\_\_} \in \Phi_a^\Theta \\ \langle \Theta,\, \Gamma,\, S \quad ,\, \texttt{raise } a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a_f = \Phi_a^\Theta(\texttt{\_\_getitem\_\_}) \end{array} \tag{13.13}$$

Setting a container item

The subscription assignment statement $e[e_i] = e_v$ sets an item e_v in the container e at key or index e_i. Like the object attribute assignment (see Section 9.4) the expressions in the container assignment are not evaluated from left to right, but the value e_v is evaluated first.

Evaluating the right hand side expression

First, the item's value e_v is evaluated.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, e[e_i] = e_v; \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|e[\,e_i\,] = \circ,\, e_v \qquad\quad \rangle \end{array} \tag{13.14}$$

Evaluating the container expression

Then, the container e is evaluated.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|e[\,e_i\,] = \circ \quad ,\, a_v \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|\circ\,[\,e_i\,] = a_v,\, e \;\rangle \end{array} \tag{13.15}$$

Evaluating the key expression

Next, the index or key e_i is evaluated.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|\circ\,[\,e_i\,] = a_v,\, a \,\rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|a[\circ] = a_v \quad ,\, e_i \rangle \end{array} \tag{13.16}$$

Calling `__setitem__`

Finally, the container's class function`__setitem__` is called to insert the new item.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|a[\circ] = a_v,\, a_i \qquad\qquad\quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, a_f([a, a_i, a_v]) \rangle & \text{if } \texttt{\_\_setitem\_\_} \in \Phi_a^\Theta \\ \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, \texttt{raise } a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a_f = \Phi_a^\Theta(\texttt{\_\_setitem\_\_}) \end{array} \tag{13.17}$$

Deleting a container item

The subscription delete statement `del` $e[e_i]$ removes an item with key or index e_i from the container e.

Evaluating the container expression

First, the container e is evaluated.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, \texttt{del } e[e_i]; \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|\texttt{del } \circ [e_i],\, e \qquad \rangle \end{array} \tag{13.18}$$

Evaluating the key expression

Then, the key or index e_i is evaluated.

$$\begin{array}{l} \langle \Theta,\, \Gamma,\, S|\texttt{del } \circ [e_i],\, a \rangle \\ \Rightarrow \langle \Theta,\, \Gamma,\, S|\texttt{del } a[\circ] \;\;,\, e_i \rangle \end{array} \tag{13.19}$$

Calling `__delitem__`

Finally, the container's class attribute `__delitem__` is called to delete the item.

$$\begin{array}{l} \quad \langle \Theta,\, \Gamma,\, S|\texttt{del } a[\circ],\, a_i \qquad\quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, a_f([a, a_i]) \rangle & \text{if } \texttt{\_\_delitem\_\_} \in \Phi_a^\Theta \\ \langle \Theta,\, \Gamma,\, S \qquad\quad ,\, \texttt{raise } a_{\texttt{TypeError}}() \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a_f = \Phi_a^\Theta(\texttt{\_\_delitem\_\_}) \end{array} \tag{13.20}$$

14 Dynamic code execution

The exec and import statements are remarkably similar. Both dynamically parse and execute a block of code, and both statements can execute a block in an isolated environment. We will first explore the simpler exec statement, and then the import statement.

14.1 The exec statement

An exec statement `exec` e_p `in` e_γ executes a program e_p in the environment e_γ. The program e_p should be a string value and the environment e_γ a dictionary.

```
exec "v␣*␣2" in { "v" : 21 }
```

Listing 14.1: Executing an important question

Evaluating the string to be executed

First, the program string e_p is evaluated.

$$\begin{aligned} &\langle \Theta, \Gamma, S \qquad\qquad\quad , \texttt{exec } e_p \texttt{ in } e_\gamma; \rangle \\ \Rightarrow &\langle \Theta, \Gamma, S|\texttt{exec } \circ \texttt{ in } e_\gamma, \ e_p \qquad\quad \rangle \end{aligned} \tag{14.1}$$

Evaluating the execution environment

Then, the environment dictionary e_γ is evaluated.

$$\begin{aligned} &\langle \Theta, \Gamma, S|\texttt{exec } \circ \texttt{ in } e_\gamma, \ a_p \rangle \\ \Rightarrow &\langle \Theta, \Gamma, S|\texttt{exec } a_p \texttt{ in } \circ, \ e_\gamma \rangle \end{aligned} \tag{14.2}$$

Execute the string

1. If a_p is a string and γ is a dictionary, we store the current environment on the stack, set the new environment and parse the program. The new environment is defined as the default environment γ_0 extended with the dictionary environment γ.

2. If either a_p is not a block, or γ is not a dictionary, we raise a `TypeError`.

$$\begin{array}{l} \left\langle \Theta,\, \Gamma \quad ,\, S | \texttt{exec}\ a_p\ \texttt{in}\ \circ ,\, \gamma \qquad\qquad\quad \right\rangle \\ \Rightarrow \begin{cases} \left\langle \Theta,\, \gamma_0 | \gamma,\, S | \Gamma \vdash \texttt{exec}\ \circ \quad ,\, \text{parse}\ \Theta(a_p) \qquad \right\rangle & \text{if } a_p :_\Theta \Theta \wedge \gamma :_\Theta \Theta \\ \left\langle \Theta,\, \Gamma \quad ,\, S \qquad\qquad\quad ,\, \texttt{raise}\ a_{\texttt{TypeError}}() \right\rangle & \text{otherwise} \end{cases} \end{array} \tag{14.3}$$

The parse function will, if the string has the correct grammar, return a block. If the parser fails however, it wil raise a `SyntaxError` exception.

Restoring the old environment

Once the program string has been executed, the old environment Γ' is reinstated and we return `None`.

$$\begin{array}{l} \left\langle \Theta,\, \Gamma\ ,\, S | \Gamma' \vdash \texttt{exec}\ \circ ,\, a \quad \right\rangle \\ \Rightarrow \left\langle \Theta,\, \Gamma',\, S \qquad\quad ,\, a_{\texttt{None}} \right\rangle \end{array} \tag{14.4}$$

Raising an exception in an execution environment

The old environment is also reinstated when an exception is raised during the execution of the program (see Chapter 10). Return, yield, break and continue statements however, can not escape the execution environment, because they cannot occur outside of their respective compound statements.

$$\begin{array}{l} \left\langle \Theta,\, \Gamma\ ,\, S | \Gamma' \vdash \texttt{exec}\ \circ ,\, \texttt{raise}\ a \right\rangle \\ \Rightarrow \left\langle \Theta,\, \Gamma',\, S \qquad\quad ,\, \texttt{raise}\ a \right\rangle \end{array} \tag{14.5}$$

14.2 The import statement

Import statements initialize modules or packages, and include them in the current environment as module objects.

Packages are directories containing an initialization module `__init__.py` and usually a number of other modules. Modules are normal Python files that are implemented by executing them. The variables that have been bound after the initialization of a module, define the attributes of the new module object. Similarly, the initialized modules and packages in a package define the attributes of its module object.

Modules and packages are initialized only once. The first time a module or package is imported, it is stored in a dictionary value at $a_{\overline{m}}$. If a module is imported a second time, it will be retrieved from the dictionary $a_{\overline{m}}$. This mechanism also prevents the built-in modules from being redefined.

An import statement `import` $\overline{x}$ imports a or package $\overline{x}$. The dot separated list of names $\overline{x}$ describes the path of a module or package, where the first names $x_{1\cdots n-1}$ are packages and the last name x_n is usually a module but can be a package.

When executing an import statement `import` $\overline{x}$ with $|\overline{x}| > 1$, each package in the path is initialized before the full path is imported. It is as if we were to execute `import` x_1, `import` $x_1.x_2$, and so forth.

Executing an import statement

The import statement is executed by placing the import frame $\Gamma \vdash$ `import` $\overline{x}$ on the stack. The import frame contains the environment Γ that is reinstated when the execution of the import statement has completed (or fails with an exception), and the path $\overline{x}$ that will be imported.

$$\begin{array}{rl} & \langle \Theta,\, \Gamma,\, S \qquad\qquad\qquad\quad ,\, \texttt{import}\ \overline{x}; \rangle \\ \Rightarrow & \langle \Theta,\, \Gamma,\, S|\Gamma \vdash \texttt{import}\ \overline{x},\, a_{\texttt{None}} \rangle \end{array} \tag{14.6}$$

This first rule is always immediately followed by Rule 14.7.

Importing modules or packages

The import frame $\Gamma \vdash$ `import` $\overline{x}.\circ.\overline{y}$ indicates that we have just imported the first part $\overline{x}$ of the import statement's path, and have to import the remaining part $\overline{y}$. The topmost environment γ_p is the environment of the module that has just been imported. The address a, which is immediately discarded, is returned by the last statement of the previously imported module.

The environment γ_p, in which the module $\overline{x}$ was executed, defines the module's attributes. This is not unlike the class definition statements (see Rule 9.1).

1. If a module or package x_m was already imported, its address is in list of previously imported modules $\overline{m}$. In this case we bind the module or package in the environment γ_p.

2. If x_m is a package (there is a directory by that name with an initialization module in it), we create a new environment a_m in which we execute the parsed initialization module. The environment a_m is bound to $\overline{y}$ in the module γ_p and stored in the list of modules $\overline{m}$.

3. If we are importing the last element of the path and it is a module, we execute the parsed module in the environment a_m. The environment a_m is bound to $\overline{y}$ in the module γ_p and stored in the list of modules $\overline{m}$.

4. Otherwise the import is invalid because a module or package could not be found, so we raise an `ImportError`.

$$
\begin{array}{l}
\quad \big\langle \Theta\ ,\ \Gamma' | \gamma_p\ ,\ S | \Gamma \vdash \texttt{import}\ \overline{x}.\circ.(y : \overline{y}),\ a \qquad\quad \big\rangle \\
\Rightarrow \begin{cases}
\big\langle \Theta'\ ,\ \overline{m}(x)\ ,\ S|f & ,\ a_{\texttt{None}} \big\rangle & \text{if } x \in \overline{m} \\
\big\langle \Theta''\ ,\ \gamma_0 | a_m,\ S|f & ,\ \text{parse } p \big\rangle & \text{if } x \notin \overline{m} \wedge x \text{ is package} \\
\big\langle \Theta''\ ,\ \gamma_0 | a_m,\ S|f & ,\ \text{parse } x.\texttt{py} \big\rangle & \text{if } x \notin \overline{m} \wedge x \text{ is module} \wedge |\overline{y}| \equiv 0 \\
\big\langle \Theta\ ,\ \Gamma\ ,\ S & ,\ \texttt{raise}\ a_{\texttt{IE}}() \big\rangle & \text{otherwise}
\end{cases} \\
\quad \textbf{where}\ a_m = \text{new address} \notin \Theta \\
\qquad\quad x = x_1/\cdots/x_n/y \\
\qquad\quad p = x/\texttt{\_\_init\_\_.py} \\
\qquad\quad \overline{m} = \Theta(a_{\overline{m}}) \\
\qquad\quad \Theta' = \Theta \oplus [\gamma_p \rightarrow \Theta(\gamma_p) \oplus [y \rightarrow \overline{m}(x)]] \\
\qquad\quad \Theta'' = \Theta \oplus [\gamma_p \rightarrow \Theta(\gamma_p) \oplus [y \rightarrow a_m] \\
\qquad\qquad\qquad\quad , a_{\overline{m}} \rightarrow \overline{m} \oplus [x \rightarrow a_m] \\
\qquad\qquad\qquad\quad , a_m \rightarrow \emptyset \\
\qquad\qquad\qquad\quad] \\
\qquad\quad a_{\texttt{IE}} = a_{\texttt{ImportError}} \\
\qquad\quad f = \Gamma \vdash \texttt{import}\ (\overline{x} +\!\!+ [y]).\circ.\overline{y}
\end{array}
\tag{14.7}
$$

Like the parser used for the exec statement (see Section 14.1), this parser returns a block when it succeeds and raises a `SyntaxError` exception when it fails. Note that we're parsing files here, not strings.

This rule applies repeatedly until either an exception is raised, or all the packages and the module have been imported. It is then followed by Rule 14.8.

Returning from the import statement

When all packages and the final module have been imported, we reinstate the environment γ and return `None`.

$$
\begin{array}{l}
\quad \big\langle \Theta,\ \Gamma' | \gamma_p,\ S | \Gamma \vdash \texttt{import}\ \overline{x}.\circ.[\,],\ a \quad \big\rangle \\
\Rightarrow \big\langle \Theta,\ \Gamma\ ,\ S \quad ,\ a_{\texttt{None}} \big\rangle
\end{array}
\tag{14.8}
$$

Raising an exception in an imported module

If an imported module raises an exception, the previous environment Γ' is reinstated and unwinding continues (see Chapter 10).

$$
\begin{array}{l}
\quad \big\langle \Theta,\ \Gamma\ ,\ S | \Gamma' \vdash \texttt{import}\ \overline{x}.\circ.\overline{y},\ \texttt{raise}\ a \big\rangle \\
\Rightarrow \big\langle \Theta,\ \Gamma',\ S \quad ,\ \texttt{raise}\ a \big\rangle
\end{array}
\tag{14.9}
$$

15 Conclusion

We have created an executable operational semantics of a significant subset of Python. The subset, called *minpy*, covers all major constructs of Python. The semantics are easy to read, insofar the formal semantics of a serious programming language can be.

15.1 Literate programming

Developing an executable operational semantics was a challenging exercise of literate programming as introduced by Knuth [8].

In contrast to Knuth's WEB system, which transparently includes a program's source code into a document, we have tried to hide the Haskell source code in this document. Had we simply included the Haskell code, it would have been perceived as just an implementation of Python in Haskell, rather than an operational semantics of Python.

The Haskell source code was formatted with lhs2TEX. This system uses formatting rules to translate Haskell to LATEX, which can be edited independently from source code that is being formatted. For example, to reveal the Haskell code, one would simply remove the formatting rules and recompile the document.

We pushed lhs2TEXto its limits. It was designed for small formatting tweaks, not to completely transform the source code as we did. The sources had to be pre-processed by a simple Perl script and stick to strict coding conventions to get the current results.

The choice of Haskell as the specification language turned out rather well. Its declarative nature and almost mathematical syntax allowed us to write Haskell code that is close to the eventual formatted formulas.

In addition, Haskell provides many static checks which prevent many small errors. For example, undeclared variables or functions (often caused by typos) are detected, and the order and number of function arguments is verified by the type system.

The type system has also given us much more confidence in the correctness of the semantics. Despite its type inference, the type system of Haskell introduces some overhead. For example, in the semantics we rely on a convention to separate addresses from other types of values: they are all described by a variable a or γ. Haskell requires us to explicitly tag these variables, e.g. `Addr a`.

Since the type system also captures side effects, it prevents any accidental introduction of a hidden state that can affect the semantics. For example, the guard expressions used to express conditional rewrite rules cannot have side effects. After all, the act of searching for the rewrite rule that applies to a specific machine state should not influence the semantics.

Hiding Haskell was not always easy. Some concepts could have been described very concisely using a number of standard library functions, but to make this document self contained, many library functions could not be used. The formatting of Haskell also limits us to a subset of Haskell.

While we wanted to hide Haskell in the document, we've tried to keep the source code as

close to the formatted rules as possible. Haskell proved quite flexible in this respect. For example, the operator $\oplus$ is the reformatted version of `<+>` in the Haskell code.

Because of the code conventions required for proper formatting and the restrictions on Haskell, the source code of the rewrite rules will not win any beauty contests (see Section 4). Nevertheless, it is almost certainly easier to read and write than any (non-executable) equivalent LaTeX code.

Overall development on the semantics is quite a satisfying experience. When changing the semantics one would typically write some tests to explore the desired semantics, then implement a solution in Haskell, and finally, when the implementation passes the tests, ensure that it is properly formatted.

The Glasgow Haskell Compiler (GHC), directly supports literate Haskell. The source files do not need to be preprocessed by a separate tool but can be fed into GHC directly. Thus, one can disregard the restrictions while experimenting and worry about the formatting later.

15.2 The interpreter

The sources of this document and some supporting code that implements a parser, pretty printer, and an interactive interpreter, can be compiled by the Haskell compiler GHC to produce an interpreter. Like the interpreter of CPython, our interpreter has both a command-line modus and the ability to interpret files.

The following example shows a session of the interpreter in its command-line modus. First, we declare the factorial function in a single function definition statement. The statement is immediately executed if parsing succeeds. Next, we call the factorial function without an argument, which results in a `TypeError` exception. Finally we call it with a correct argument and exit the interpreter.

```
gideon@gideon-desktop:~$ minpy
>>> def fac(x) :
...     if x <= 0 :
...         return 1
...     else :
...         return fac(x-1) * x
...
>>> fac()
TypeError
>>> fac(4)
24
>>> exiting minpy
```

Without a standard library and a foreign function interface, the interpreter cannot be used for many serious applications. However, as a platform of experimentation and testing, the interpreter was invaluable.

The interpreter includes a tracer that has proven to be very helpful when debugging the rewrite rules. The tracer prints the changes of the abstract machine state.

The following interpreter session shows how the tracer is used. The tracer is first enabled by calling `enableTrace()` of the `sys` module. After a simple assignment statement the trace

shows how the value 1 is stored at address 1006, and the environment with address 22 is updated (in `state_5`) with a new mapping.

The interpreter always prints the result of the last statement if the result is not `None`. So when we execute the expression statement a, the value is retrieved (states 7 and 8) and printed (states 10 through 15).

```
>>> import sys
>>> sys.enableTrace()
state_1 = <heap_0, []:|:29:|:22, [], 0>
>>> a = 1
state_3 = <heap_2, envs_2, stack_2:|:AssFrame "a", 1>
state_4 = <heap_3<+>1006 |-> IntValue 1, envs_2, stack_3, 1006>
state_5 = <heap_4<+>22 |-> ModValue (fromList [("a",1006),("sys",25)]),
[]:|:29:|:22, stack_2, 0>
>>> a
state_7 = <heap_6, envs_6, stack_2, a>
state_8 = <heap_7, []:|:29:|:22, [], 1006>
state_10 = <heap_9, envs_9, stack_9:|:PrintFrame1, $1006>
state_11 = <heap_10, envs_9, stack_10, 1006>
state_12 = <heap_11, envs_9, stack_9:|:PrintFrame2, int.__str__:: (7)
-> ?([1006])>
state_13 = <heap_12, envs_9, stack_12, int.__str__([1006])>
state_14 = <heap_13<+>1007 |-> StringValue "1", envs_9, stack_12, 1007>
1
state_15 = <heap_14, envs_9, stack_9, 0>
```

Traces usually include a lot of addresses that are hard to keep track of. We implemented address expressions that allow us to handle addresses as first class object in the language for debugging. For example, if we had forgotten what the address 1006 pointed to we can retrieve it by entering the expression statement `$1006`.

```
>>> $1006
1
>>> exiting minpy
```

The interpreter does not only help when debugging but also provides insight in the semantics. Some aspects of the operational semantics 'emerge' from a number of rules. Judging where these emergent semantics occur and exploring their consequences, is often difficult. The interpreter allows us to explore these issues using examples.

For example, the interaction of the try-finally statement with the return statement (see Section 10.3) has some unexpected consequences for the semantics. It is quite hard to infer the semantics of this exceptional case from the rules. Running an example in the interpreter however, immediately reveals the emergent semantics.

Our interpreter runs considerably slower than CPython. This is no surprise. We paid little attention to the performance of the interpreter.

It should be noted though, that the difference is not a simple constant factor. The heap is implemented with a balanced binary tree, whereas CPython uses a direct mapping of memory

addresses to values. For a simple address lookup, the former has a complexity of $O(log(n))$, whereas the latter has a constant complexity.

Due to the absence of any garbage collector in the interpreter our interpreter uses much more memory than CPython. Indeed, it will be hard to test any memory intensive programs and even a simple program such as `while True: pass` will eventually crash due to a lack of memory.

15.3 The test suite

As mentioned before, our executable operational semantics is testable: we can execute programs and verify whether the *minpy* interpreter gives the same answer as the CPython interpreter.

The CPython implementation is developed with a suite of regression tests. The regression tests could unfortunately not be used to test *minpy* because it relies on many Python features and libraries that are not available in *minpy*.

Fortunately the efforts to reverse engineer the semantics of CPython provided an excellent opportunity to write new tests. The resulting test suite consists of 134 tests and is completely written in *minpy*, including the test driver.

Beside testing the basic behaviour of constructs, the tests verify aspects such as control flow, order of evaluation, type/class requirements, and variable scoping. Each test covers a single aspect of a single construct.

We have been able to run the test suite for a number of implementations of Python (see Table 15.1). Testing other implementations improves confidence in the tested implementations.

Judging from the low number of failed tests across implementations, the other implementations seem to implement the same language, and conversely, *minpy* and the test-suite seem to capture a common subset of Python. Hopefully *minpy* can converge on the essential core of Python.

Implementations	**Failed**
CPython 2.5.2 (reference)	1
minpy	13
PyPy 1.0.0 build 41081	3
Iron Python 1.1.1 on .net 2.0.50727.42	15
Jython 2.2.1 on java1.6.0_0	17

Table 15.1: Number of failed tests for various Python implementations.

Since CPython is the reference implementation one would expect no errors. There is, however, one test that crashes the Python interpreter completely. The developers of CPython acknowledged that this was a bug and immediately solved the problem. None of the other implementations crashed in the same situation.

Our interpreter fails 13 tests, all of which are by definition errors in our semantics. Most of the errors are considered missing features. For example, lists in *minpy* raise an exception when a negative index is used. CPython on the other hand, treats negative indices as normal

indices on the reversed list with an offset of 1. Thus in CPython $[1, 2][-1]$ evaluates to the value 2 but raises an exception in *minpy*.

Of course there are many more errors in *minpy* than have been tested or even discovered. Some of the known issues are listed in Section 15.4.

Interpreting the test results of other Python implementations is difficult. Some differences are intentional and can therefore not be considered bugs. Whether or not a failed test indicates a bug is a matter of opinion. Nevertheless we tried to interpret the results somewhat:

The PyPy interpreter failed three tests due to incompatibilities only; none of the failures indicated bugs. Iron Python failed quite a number of tests, but only five of those can be considered bugs. Of the failed tests on Jython, only three can be considered bugs.

15.4 Future work

There is an enormous body of work that can be based on this operational semantics. First of all, *minpy* can be expanded to cover all, or at least a more complete subset of Python.

The grammar of *minpy* and therefore the specification, is much simpler than Python's syntax (see Chapter 3). Most significantly, list and generator comprehensions, keyword arguments and other advanced function parameters, the with statement, and the global statement are missing.

Many optional elements and other syntactic sugar are also missing in *minpy*. This makes the semantics less verbose and allows us to concentrate on the most interesting aspects of Python. Nevertheless, for the sake of completeness, they should eventually be specified.

It should be noted that many constructs that could be considered syntactic sugar, actually have semantics that cannot completely be described by a simple transformation. For example, the augmented assignment $x \mathrel{+}= 1$ is not equivalent to $x = x+1$: in the former case Python would call the class function `__iadd__`, but for the latter it would call `__setattr__` and `__add__`.

We are confident that all constructs of Python that are missing from *minpy* can be added without significant changes to the existing semantic rules. The structure of the abstract machine will not have to be changed, and the framework used to describe the rules will not need significant extensions.

Secondly, there are some errors in the current semantics that should be corrected. The test-suite pointed out some errors in the semantics of class initialization. Although we do use the class of a class –the metaclass– to construct classes, the mechanics of metaclasses have not been covered completely (see Section 9.1, specifically Rule 9.3).

Most of the built-in types have been specified to make the interpreter usable (see Chapter A). But the current semantics of *minpy*'s built-in types are not conform to Python. For example, *minpy* does not support coercion, and the dictionary class only accepts strings as keys.

Thirdly, the scope of the semantics (see Section 1.1) can be expanded to encompass more of Python's aspects:

- Garbage collection and its interaction with the operational semantics can be explored an formalised.
- Concurrent programming, notorious for its complexity, can be formalised.
- The interaction of Python with the outside world –the foreign function interface (FFI)– can be specified.

- The various reflective features can be formalised. It seems that most of these features can be specified fairly easily. For example, the `__dict__` attribute that exposes the attribute mappings of an object, would not require big changes.

For completeness one might consider to add so called old-style classes as well. It might be more productive to update the entire specification to a newer version of Python, that will does not support old style classes.

Fourthly, the interpreter, which will benefit from all improvements of the semantics, can be improved. Currently the error feedback is minimal. It would be very useful if raised exceptions also print a stack trace.

The performance of the interpreter can probably be improved, but it is doubtful it will ever achieve performance comparable to CPython without compromising the legibility of the semantics.

The interpreter could also be used as a basis for various other tools. The current basic tracing functionality could be extended to a fully fledged debugger for example.

Finally, our work opens many new avenues of research. The lack of a formal semantics for Python prevented any formal work on Python. Now however we can pursue various new directions of research:

- One can prove properties of Python with respect to our semantics. For example, we would very much like to prove that the execution of any possible Python program will never end in a state, other than the final state, that cannot be rewritten. Such a proof would have prevented the bug that we found in CPython.

- New tools for and analyses of Python programs can be developed more easily. For example, a tool that type-checks a Python program could be developed. Perhaps some properties of the type system could even be proven correct.

- Language extensions can be developed within our framework. Resulting in both a formal description of the extension and a proof of concept interpreter.

A Builtin values

A.1 The base objects `object` and `type`

$$
\begin{array}{ll}
o_{\texttt{object}} = \cdot & \\
\quad [\,\texttt{\_\_getattribute\_\_} & \rightarrow \texttt{object.\_\_getattribute\_\_} \\
\quad ,\,\texttt{\_\_setattr\_\_} & \rightarrow \texttt{object.\_\_setattr\_\_} \\
\quad ,\,\texttt{\_\_delattr\_\_} & \rightarrow \texttt{object.\_\_delattr\_\_} \\
\quad ,\,\texttt{\_\_new\_\_} & \rightarrow \texttt{object.\_\_new\_\_} \\
\quad ,\,\texttt{\_\_init\_\_} & \rightarrow \texttt{object.\_\_init\_\_} \\
\quad ,\,\texttt{\_\_str\_\_} & \rightarrow \texttt{object.\_\_str\_\_} \\
\quad ,\,\texttt{\_\_bases\_\_} & \rightarrow a_{()} \\
\quad ,\,\texttt{\_\_class\_\_} & \rightarrow a_{\texttt{type}} \\
\quad] &
\end{array}
$$

$$
\begin{array}{ll}
o_{\texttt{type}} = \cdot & \\
\quad [\,\texttt{\_\_getattribute\_\_} & \rightarrow \texttt{type.\_\_getattribute\_\_} \\
\quad ,\,\texttt{\_\_bases\_\_} & \rightarrow a_{(\texttt{object},)} \\
\quad ,\,\texttt{\_\_class\_\_} & \rightarrow a_{\texttt{type}} \\
\quad ,\,\texttt{\_\_new\_\_} & \rightarrow \texttt{type.\_\_new\_\_} \\
\quad ,\,\texttt{\_\_call\_\_} & \rightarrow \texttt{type.\_\_call\_\_} \\
\quad ,\,\texttt{\_\_str\_\_} & \rightarrow \texttt{type.\_\_str\_\_} \\
\quad] &
\end{array}
$$

The default string representation

$$
\begin{array}{l}
\quad \big\langle \Theta \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , \Gamma, S, \texttt{object.\_\_str\_\_}([\,a_{\texttt{self}}\,]) \big\rangle \\
\Rightarrow \big\langle \Theta \oplus [\,a \rightarrow (\texttt{"<object at "} +\!\!+\ \textit{show}\ a_{\texttt{self}} +\!\!+ \texttt{">"})], \Gamma, S, a \big\rangle \\
\textbf{where}\ a = \text{new address} \notin \Theta
\end{array}
\tag{A.1}
$$

Default string representation for classes

$$
\begin{array}{l}
\quad \big\langle \Theta \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , \Gamma, S, \texttt{type.\_\_str\_\_}([\,a_{\texttt{self}}\,]) \big\rangle \\
\Rightarrow \big\langle \Theta \oplus [\,a \rightarrow (\texttt{"<class at "} +\!\!+\ \textit{show}\ a_{\texttt{self}} +\!\!+ \texttt{">"})], \Gamma, S, a \big\rangle \\
\textbf{where}\ a = \text{new address} \notin \Theta
\end{array}
\tag{A.2}
$$

A.2 The function and generator

$$
\begin{aligned}
&o_{\texttt{generator}} = \cdot \\
&\quad [\texttt{\_\_class\_\_} \rightarrow a_{\texttt{type}} \\
&\quad , \texttt{\_\_bases\_\_} \rightarrow a_{(\texttt{object},)} \\
&\quad , \texttt{\_\_next\_\_} \rightarrow \texttt{generator.\_\_next\_\_} \\
&\quad , \texttt{\_\_send\_\_} \rightarrow \texttt{generator.\_\_send\_\_} \\
&\quad , \texttt{\_\_iter\_\_} \rightarrow \texttt{generator.\_\_iter\_\_} \\
&\quad]
\end{aligned}
$$

$$
\begin{aligned}
&o_{\texttt{function}} = \cdot \\
&\quad [\texttt{\_\_class\_\_} \rightarrow a_{\texttt{type}} \\
&\quad , \texttt{\_\_bases\_\_} \rightarrow a_{(\texttt{object},)} \\
&\quad , \texttt{\_\_call\_\_} \rightarrow \texttt{function.\_\_call\_\_} \\
&\quad]
\end{aligned}
$$

A function as a callable object

$$
\begin{aligned}
&\left\langle \Theta,\, \Gamma,\, S,\, \texttt{function.\_\_call\_\_}([a]) \right\rangle \\
\Rightarrow &\left\langle \Theta,\, \Gamma,\, S,\, a \right\rangle
\end{aligned}
\tag{A.3}
$$

The generator as an iterable

$$
\begin{aligned}
&\left\langle \Theta,\, \gamma,\, S,\, \texttt{generator.\_\_iter\_\_}([a_{\texttt{self}}]) \right\rangle \\
\Rightarrow &\left\langle \Theta,\, \gamma,\, S,\, a_{\texttt{self}} \right\rangle
\end{aligned}
\tag{A.4}
$$

A.3 Integer values

The integer class

$$
\begin{aligned}
&o_{\texttt{int}} = \cdot \\
&\quad [\texttt{\_\_class\_\_} \rightarrow a_{\texttt{type}} \\
&\quad , \texttt{\_\_bases\_\_} \rightarrow a_{(\texttt{object},)} \\
&\quad , \texttt{\_\_new\_\_} \rightarrow \texttt{int.\_\_new\_\_} \\
&\quad , \texttt{\_\_add\_\_} \rightarrow \texttt{int.\_\_add\_\_} : [a_{\texttt{int}}, a_{\texttt{int}}] \\
&\quad , \texttt{\_\_sub\_\_} \rightarrow \texttt{int.\_\_sub\_\_} : [a_{\texttt{int}}, a_{\texttt{int}}] \\
&\quad , \texttt{\_\_mul\_\_} \rightarrow \texttt{int.\_\_mul\_\_} : [a_{\texttt{int}}, a_{\texttt{int}}] \\
&\quad , \texttt{\_\_div\_\_} \rightarrow \texttt{int.\_\_div\_\_} : [a_{\texttt{int}}, a_{\texttt{int}}] \\
&\quad , \texttt{\_\_divmod\_\_} \rightarrow \texttt{int.\_\_divmod\_\_} : [a_{\texttt{int}}, a_{\texttt{int}}] \\
&\quad , \texttt{\_\_floordiv\_\_} \rightarrow \texttt{int.\_\_floordiv\_\_} : [a_{\texttt{int}}, a_{\texttt{int}}] \\
&\quad , \texttt{\_\_pow\_\_} \rightarrow \texttt{int.\_\_pow\_\_} : [a_{\texttt{int}}, a_{\texttt{int}}]
\end{aligned}
$$

$$
\begin{array}{ll}
, \texttt{\_\_neq\_\_} & \rightarrow \texttt{int.\_\_neq\_\_} : [\, a_{\texttt{int}}, a_{\texttt{int}} \,] \\
, \texttt{\_\_eq\_\_} & \rightarrow \texttt{int.\_\_eq\_\_} : [\, a_{\texttt{int}}, a_{\texttt{int}} \,] \\
, \texttt{\_\_lt\_\_} & \rightarrow \texttt{int.\_\_lt\_\_} : [\, a_{\texttt{int}}, a_{\texttt{int}} \,] \\
, \texttt{\_\_le\_\_} & \rightarrow \texttt{int.\_\_le\_\_} : [\, a_{\texttt{int}}, a_{\texttt{int}} \,] \\
, \texttt{\_\_gt\_\_} & \rightarrow \texttt{int.\_\_gt\_\_} : [\, a_{\texttt{int}}, a_{\texttt{int}} \,] \\
, \texttt{\_\_ge\_\_} & \rightarrow \texttt{int.\_\_ge\_\_} : [\, a_{\texttt{int}}, a_{\texttt{int}} \,] \\
, \texttt{\_\_str\_\_} & \rightarrow \texttt{int.\_\_str\_\_} : [\, a_{\texttt{int}} \,] \\
\,]
\end{array}
$$

Literal integers

Evaluating an integer literal

$$
\begin{array}{l}
\quad \langle \Theta \qquad\qquad , \Gamma, S, i \rangle \\
\Rightarrow \langle \Theta \oplus [a \rightarrow i], \Gamma, S, a \rangle \\
\textbf{where } a = \text{new address} \notin \Theta
\end{array}
\tag{A.5}
$$

Integer primitives

The addition primitive

$$
\begin{array}{l}
\quad \langle \Theta, \Gamma, S, \texttt{int.\_\_add\_\_}([\, a_x, a_y \,]) \rangle \\
\Rightarrow \langle \Theta, \Gamma, S, (\Theta(a_x) + \Theta(a_y)) \quad \rangle
\end{array}
\tag{A.6}
$$

The equals primitive

$$
\begin{array}{l}
\quad \langle \Theta, \Gamma, S, \texttt{int.\_\_eq\_\_}([\, a_x, a_y \,]) \rangle \\
\Rightarrow \langle \Theta, \Gamma, S, \Theta(a_x) \equiv \Theta(a_y) \quad \rangle
\end{array}
\tag{A.7}
$$

The constructor primitive

$$
\begin{array}{l}
\quad \langle \Theta \qquad\qquad , \Gamma, S, \texttt{int.\_\_new\_\_}([\, a_c \,]) \rangle \\
\Rightarrow \langle \Theta \oplus [a \rightarrow o], \Gamma, S, a \qquad\qquad \rangle \\
\textbf{where } a = \text{new address} \notin \Theta \\
\qquad\quad o = ([\texttt{\_\_class\_\_} \rightarrow a_c], 0)
\end{array}
\tag{A.8}
$$

The string representation

$$
\begin{array}{l}
\quad \langle \Theta \qquad\qquad\qquad , \Gamma, S, \texttt{int.\_\_str\_\_}([\, a_o \,]) \rangle \\
\Rightarrow \langle \Theta \oplus [a \rightarrow (\mathit{show}\ \Theta(a_o))], \Gamma, S, a \qquad\qquad\qquad \rangle \\
\textbf{where } a = \text{new address} \notin \Theta
\end{array}
\tag{A.9}
$$

A.4 Booleans

The boolean class

$$
\begin{array}{l}
\texttt{bool} = \cdot \\
\quad [\texttt{\_\_class\_\_} \quad \rightarrow a_{\texttt{type}} \\
\quad , \texttt{\_\_bases\_\_} \quad \rightarrow a_{(\texttt{object},)} \\
\quad , \texttt{\_\_nonzero\_\_} \rightarrow \texttt{bool.\_\_nonzero\_\_} : [a_{\texttt{bool}}] \\
\quad , \texttt{\_\_str\_\_} \quad \rightarrow \texttt{bool.\_\_str\_\_} : [a_{\texttt{bool}}] \\
\quad]
\end{array}
$$

Literal booleans

Evaluating a boolean literal

$$
\begin{array}{l}
\quad \langle \Theta \qquad\qquad , \gamma, S, b \rangle \\
\Rightarrow \langle \Theta \oplus [a \rightarrow (b)], \gamma, S, a \rangle \\
\textbf{where } a = \text{new address} \notin \Theta
\end{array}
\tag{A.10}
$$

Boolean primitives

The boolean to boolean conversion primitive

$$
\begin{array}{l}
\quad \langle \Theta, \gamma, S, \texttt{bool.\_\_nonzero\_\_}([a]) \rangle \\
\Rightarrow \langle \Theta, \gamma, S, a \qquad\qquad\qquad\quad \rangle
\end{array}
\tag{A.11}
$$

The boolean to string conversion primitive

$$
\begin{array}{l}
\quad \langle \Theta, \gamma, S, \texttt{bool.\_\_str\_\_}([a]) \rangle \\
\Rightarrow \begin{cases} \langle \Theta, \gamma, S, \texttt{"True"} \quad\quad\quad \rangle & \text{if } \Theta(a) \\ \langle \Theta, \gamma, S, \texttt{"False"} \quad\quad\quad \rangle & \text{otherwise} \end{cases}
\end{array}
\tag{A.12}
$$

A.5 Strings

The string class

$$
\begin{array}{l}
o_{string} = \cdot \\
\quad [\texttt{\_\_class\_\_} \quad \rightarrow a_{\texttt{type}} \\
\quad , \texttt{\_\_bases\_\_} \quad \rightarrow a_{(\texttt{object},)} \\
\quad , \texttt{\_\_new\_\_} \quad \rightarrow \texttt{str.\_\_new\_\_} \\
\quad , \texttt{\_\_add\_\_} \quad \rightarrow \texttt{str.\_\_add\_\_} : [a_{\texttt{str}}, a_{\texttt{str}}] \\
\quad , \texttt{\_\_mul\_\_} \quad \rightarrow \texttt{str.\_\_mul\_\_} : [a_{\texttt{str}}, a_{\texttt{int}}]
\end{array}
$$

$$
\begin{array}{ll}
, \texttt{\_\_len\_\_} & \rightarrow \texttt{str.\_\_len\_\_} : [a_{\texttt{str}}] \\
, \texttt{\_\_eq\_\_} & \rightarrow \texttt{str.\_\_eq\_\_} : [a_{\texttt{str}}, a_{\texttt{str}}] \\
, \texttt{\_\_getitem\_\_} & \rightarrow \texttt{str.\_\_getitem\_\_} : [a_{\texttt{str}}, a_{\texttt{int}}] \\
, \texttt{\_\_str\_\_} & \rightarrow \texttt{str.\_\_str\_\_} : [a_{\texttt{str}}] \\
, \texttt{"append"} & \rightarrow \texttt{str.\_\_add\_\_} : [a_{\texttt{str}}, a_{\texttt{str}}] \\
] &
\end{array}
$$

Literal strings

Evaluating a string literal

$$
\begin{array}{l}
\quad \langle \Theta \quad , \gamma, S, t \rangle \\
\Rightarrow \langle \Theta \oplus [a \rightarrow t], \gamma, S, a \rangle \\
\textbf{where } a = \text{new address} \notin \Theta
\end{array}
\tag{A.13}
$$

String primitives

The string constructor primitive

$$
\begin{array}{l}
\quad \langle \Theta \quad , \gamma, S, \texttt{str.\_\_new\_\_}([a_c]) \rangle \\
\Rightarrow \langle \Theta \oplus [a \rightarrow o], \gamma, S, a \quad \rangle \\
\textbf{where } a = \text{new address} \notin \Theta \\
\qquad o = ([\texttt{\_\_class\_\_} \rightarrow a_c], [\,])
\end{array}
\tag{A.14}
$$

The string constructor primitive with starting string

$$
\begin{array}{l}
\quad \langle \Theta \quad , \gamma, S, \texttt{str.\_\_new\_\_}([a_c, a_{\texttt{str}}]) \rangle \\
\Rightarrow \langle \Theta \oplus [a \rightarrow o], \gamma, S, a \quad \rangle \\
\textbf{where } a = \text{new address} \notin \Theta \\
\qquad o = ([\texttt{\_\_class\_\_} \rightarrow a_c], \Theta(a_{\texttt{str}}))
\end{array}
\tag{A.15}
$$

The length of string primitive

$$
\begin{array}{l}
\quad \langle \Theta, \gamma, S, \texttt{str.\_\_len\_\_}([a]) \rangle \\
\Rightarrow \langle \Theta, \gamma, S, (|\Theta(a)|) \quad \rangle
\end{array}
\tag{A.16}
$$

The string addition operator primitive

$$
\begin{array}{l}
\quad \langle \Theta \quad , \gamma, S, \texttt{str.\_\_add\_\_}([a_{\texttt{self}}, a_{\texttt{other}}]) \rangle \\
\Rightarrow \langle \Theta \oplus [a \rightarrow s], \gamma, S, a \quad \rangle \\
\textbf{where } s = (\Theta(a_{\texttt{self}}) +\!\!+ \Theta(a_{\texttt{other}})) \\
\qquad a = \text{new address} \notin \Theta
\end{array}
\tag{A.17}
$$

The string equals primitive

$$\begin{array}{l} \langle \Theta,\, \gamma,\, S,\, \texttt{str.\_\_eq\_\_}([a_x, a_y]) \rangle \\ \Rightarrow \langle \Theta,\, \gamma,\, S,\, \Theta(a_x) \equiv \Theta(a_y) \rangle \end{array} \tag{A.18}$$

The string get character at index primitive

$$\begin{array}{l} \langle \Theta \quad , \gamma,\, S,\, \texttt{str.\_\_getitem\_\_}([a_{\texttt{self}}, a_i]) \rangle \\ \Rightarrow \begin{cases} \langle \Theta \oplus [a \to [s \mathbin{!!} i]],\, \gamma,\, S,\, a \rangle & \text{if } i < |s| \\ \langle \Theta \quad , \gamma,\, S,\, \texttt{raise } a_{\texttt{IndexError}}() \rangle & \text{otherwise} \end{cases} \\ \textbf{where } i = \Theta(a_i) \\ \qquad s = \Theta(a_{\texttt{self}}) \\ \qquad a = \text{new address} \notin \Theta \end{array} \tag{A.19}$$

The string to string conversion primitive

$$\begin{array}{l} \langle \Theta \quad , \gamma,\, S,\, \texttt{str.\_\_str\_\_}([a_o]) \rangle \\ \Rightarrow \langle \Theta \oplus [a \to \Theta(a_o)],\, \gamma,\, S,\, a \rangle \\ \textbf{where } a = \text{new address} \notin \Theta \end{array} \tag{A.20}$$

A.6 Lists

The list class

$$\begin{array}{l} o_{\texttt{list}} = \cdot \\ \quad [\texttt{\_\_class\_\_} \to a_{\texttt{type}} \\ \quad ,\texttt{\_\_bases\_\_} \to a_{(\texttt{object},)} \\ \quad ,\texttt{\_\_delitem\_\_} \to \texttt{list.\_\_delitem\_\_} : [a_{\texttt{list}}, a_{\texttt{int}}] \\ \quad ,\texttt{\_\_getitem\_\_} \to \texttt{list.\_\_getitem\_\_} : [a_{\texttt{list}}, a_{\texttt{int}}] \\ \quad ,\texttt{\_\_setitem\_\_} \to \texttt{list.\_\_setitem\_\_} : [a_{\texttt{list}}, a_{\texttt{int}}] \\ \quad ,\texttt{\_\_new\_\_} \to \texttt{list.\_\_new\_\_} \\ \quad ,\texttt{\_\_add\_\_} \to \texttt{list.\_\_add\_\_} \\ \quad ,\texttt{\_\_len\_\_} \to \texttt{list.\_\_len\_\_} \\ \quad ,\texttt{\_\_iter\_\_} \to \texttt{list.\_\_iter\_\_} \\ \quad ,\texttt{append} \to \texttt{list.append} \\ \quad] \end{array}$$

Literal strings

Evaluating a list literal

$$\begin{array}{l} \langle \Theta \quad , \gamma,\, S \quad , [\overline{e}] \rangle \\ \Rightarrow \begin{cases} \langle \Theta \quad , \gamma,\, S | [\circ, es'],\, e \rangle & \text{if } \overline{e} = (e : es') \\ \langle \Theta \oplus [a \to []],\, \gamma,\, S \quad , a \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a = \text{new address} \notin \Theta \end{array} \tag{A.21}$$

Evaluating the elements of a list literal

$$\begin{array}{l} \quad \langle \Theta \quad , \gamma, S|[\overline{a}, \circ, \overline{e}] \quad , a \rangle \\ \Rightarrow \begin{cases} \langle \Theta \quad , \gamma, S|[\overline{a} +\!\!+ [a], \circ, es'], e \rangle & \text{if } \overline{e} = (e : es') \\ \langle \Theta \oplus [a' \to [\overline{a} +\!\!+ [a]]], \gamma, S \quad , a' \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a' = \text{new address} \notin \Theta \end{array} \tag{A.22}$$

List primitives

The list constructor primitive

$$\begin{array}{l} \quad \langle \Theta \quad , \gamma, S, \texttt{list.\_\_new\_\_}([a_c]) \rangle \\ \Rightarrow \langle \Theta \oplus [a \to o], \gamma, S, a \rangle \\ \textbf{where } a = \text{new address} \notin \Theta \\ \qquad o = ([\texttt{\_\_class\_\_} \to a_c], [\,]) \end{array} \tag{A.23}$$

The length of list primitive

$$\begin{array}{l} \quad \langle \Theta, \gamma, S, \texttt{list.\_\_len\_\_}([a]) \rangle \\ \Rightarrow \langle \Theta, \gamma, S, (|\Theta(a)|) \rangle \end{array} \tag{A.24}$$

The get item at index primitive

$$\begin{array}{l} \quad \langle \Theta, \gamma, S, \texttt{list.\_\_getitem\_\_}([a_{\texttt{self}}, a_i]) \rangle \\ \Rightarrow \begin{cases} \langle \Theta, \gamma, S, \overline{a} \,!!\, i \rangle & \text{if } i < |\overline{a}| \\ \langle \Theta, \gamma, S, \texttt{raise } a_{\texttt{IndexError}}() \rangle & \text{otherwise} \end{cases} \\ \textbf{where } i = \Theta(a_i) \\ \qquad \overline{a} = \Theta(a_{\texttt{self}}) \end{array} \tag{A.25}$$

The set item at index primitive

$$\begin{array}{l} \quad \langle \Theta \quad , \gamma, S, \texttt{list.\_\_setitem\_\_}([a_{\texttt{self}}, a_i, a]) \rangle \\ \Rightarrow \begin{cases} \langle \Theta \oplus [a_{\texttt{self}} \to \overline{a}'], \gamma, S, a_{\texttt{None}} \rangle & \text{if } i < |\overline{a}| \\ \langle \Theta \quad , \gamma, S, \texttt{raise } a_{\texttt{IndexError}}() \rangle & \text{otherwise} \end{cases} \\ \textbf{where } i \;\; = \Theta(a_i) \\ \qquad \overline{a} \;\; = \Theta(a_{\texttt{self}}) \\ \qquad \overline{a}' = [\cdots, a_{i-1}, a, a_{i+1}, \cdots] \end{array} \tag{A.26}$$

The delete item at index primitive

$$\begin{array}{l} \quad \langle \Theta \qquad\qquad , \gamma, S, \texttt{list.\_\_delitem\_\_}([a_{\texttt{self}}, a_i]) \rangle \\ \Rightarrow \begin{cases} \langle \Theta \oplus [a_{\texttt{self}} \to \overline{a}'], \gamma, S, a_{\texttt{None}} \rangle & \text{if } i < |\overline{a}| \\ \langle \Theta \qquad\qquad , \gamma, S, \texttt{raise } a_{\texttt{IndexError}}() \rangle & \text{otherwise} \end{cases} \\ \textbf{where } i = \Theta(a_i) \\ \qquad\quad \overline{a} = \Theta(a_{\texttt{self}}) \\ \qquad\quad \overline{a}' = [\cdots, a_{i-1}, a_{i+1}, \cdots] \end{array} \tag{A.27}$$

The append to list primitive

$$\begin{array}{l} \quad \langle \Theta \qquad\qquad , \gamma, S, \texttt{list.append}([a_{\texttt{self}}, a]) \rangle \\ \Rightarrow \langle \Theta \oplus [a_{\texttt{self}} \to \overline{a}], \gamma, S, a_{\texttt{None}} \rangle \\ \textbf{where } \overline{a} = [\Theta(a_{\texttt{self}}) +\!\!+ [a]] \end{array} \tag{A.28}$$

The list addition operator primitive

$$\begin{array}{l} \quad \langle \Theta \qquad\qquad , \gamma, S, \texttt{list.\_\_add\_\_}([a_{\texttt{self}}, a_{\texttt{other}}]) \rangle \\ \Rightarrow \langle \Theta \oplus [a \to \overline{a}], \gamma, S, a \rangle \\ \textbf{where } \overline{a} = [\Theta(a_{\texttt{self}}) +\!\!+ \Theta(a_{\texttt{other}})] \\ \qquad\quad a = \text{new address} \notin \Theta \end{array} \tag{A.29}$$

A.7 Tuples

A.8 The tuple class

$$\begin{array}{l} o_{\texttt{tuple}} = \cdot \\ \quad [\texttt{\_\_class\_\_} \quad \to a_{\texttt{type}} \\ \quad , \texttt{\_\_bases\_\_} \quad \to a_{(\texttt{object},)} \\ \quad , \texttt{\_\_getitem\_\_} \to \texttt{tuple.\_\_getitem\_\_} : [a_{\texttt{tuple}}, a_{\texttt{int}}] \\ \quad , \texttt{\_\_new\_\_} \quad \to \texttt{tuple.\_\_new\_\_} \\ \quad , \texttt{\_\_len\_\_} \quad \to \texttt{tuple.\_\_len\_\_} \\ \quad] \end{array}$$

Literal tuples

Evaluating a tuple literal

$$\begin{array}{l} \quad \langle \Theta \qquad\qquad , \gamma, S \qquad , (\overline{e}) \rangle \\ \Rightarrow \begin{cases} \langle \Theta \qquad\qquad , \gamma, S|(\circ, es'), e \rangle & \text{if } \overline{e} = (e : es') \\ \langle \Theta \oplus [a \to (,)], \gamma, S \qquad , a \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a = \text{new address} \notin \Theta \end{array} \tag{A.30}$$

Evaluating the elements of a tuple literal

$$\begin{array}{l} \quad \langle \Theta \quad , \gamma, S|(\overline{a}, \circ, \overline{e}) \quad , a \rangle \\ \Rightarrow \begin{cases} \langle \Theta \quad , \gamma, S|(\overline{a} +\!\!+ [a], \circ, \overline{e}'), e \rangle & \text{if } \overline{e} = (e : \overline{e}') \\ \langle \Theta \oplus [a' \to (\overline{a} +\!\!+ [a])], \gamma, S \quad , a' \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a' = \text{new address} \notin \Theta \end{array} \tag{A.31}$$

Tuple primitives

The tuple constructor primitive

$$\begin{array}{l} \quad \langle \Theta \quad , \gamma, S, \texttt{tuple.\_\_new\_\_}([a_c]) \rangle \\ \Rightarrow \langle \Theta \oplus [a \to o], \gamma, S, a \rangle \\ \textbf{where } a = \text{new address} \notin \Theta \\ \qquad o = ([\texttt{\_\_class\_\_} \to a_c], ([])) \end{array} \tag{A.32}$$

The length of tuple primitive

$$\begin{array}{l} \quad \langle \Theta, \gamma, S, \texttt{tuple.\_\_len\_\_}([a]) \rangle \\ \Rightarrow \langle \Theta, \gamma, S, (|\Theta(a)|) \rangle \end{array} \tag{A.33}$$

The get item at index primitive

$$\begin{array}{l} \quad \langle \Theta, \gamma, S, \texttt{tuple.\_\_getitem\_\_}([a_{\texttt{self}}, a_i]) \rangle \\ \Rightarrow \begin{cases} \langle \Theta, \gamma, S, \overline{a} \mathbin{!!} i \rangle & \text{if } i < |\overline{a}| \\ \langle \Theta, \gamma, S, \texttt{raise } a_{\texttt{IndexError}} \rangle & \text{otherwise} \end{cases} \\ \textbf{where } i = \Theta(a_i) \\ \qquad \overline{a} = \Theta(a_{\texttt{self}}) \end{array} \tag{A.34}$$

A.9 Dictionaries

Dictionary class

$$\begin{array}{l} o_{\texttt{dict}} = \cdot \\ \quad [\texttt{\_\_class\_\_} \quad \to a_{\texttt{type}} \\ \quad , \texttt{\_\_bases\_\_} \quad \to a_{(\texttt{object},)} \\ \quad , \texttt{\_\_getitem\_\_} \to \texttt{dict.\_\_getitem\_\_} : [a_{\texttt{dict}}, a_{\texttt{str}}] \\ \quad , \texttt{\_\_setitem\_\_} \to \texttt{dict.\_\_setitem\_\_} : [a_{\texttt{dict}}, a_{\texttt{str}}, a_{\texttt{object}}] \\ \quad , \texttt{\_\_delitem\_\_} \to \texttt{dict.\_\_delitem\_\_} : [a_{\texttt{dict}}, a_{\texttt{str}}] \\ \quad , \texttt{\_\_new\_\_} \quad \to \texttt{dict.\_\_new\_\_} \\ \quad] \end{array}$$

Dictionary literals

Evaluating a dictionary literal

$$\begin{array}{l} \langle \Theta \quad , \gamma, S \quad , \overline{\overline{e}} \quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta \quad , \gamma, S|\{\emptyset, \circ : \overline{\overline{e}}_{1;2}, tail\ \overline{\overline{e}}\}, \overline{\overline{e}}_{1;1} \rangle & \text{if } |\overline{\overline{e}}| > 0 \\ \langle \Theta \oplus [a \to \emptyset], \gamma, S \quad , a \quad \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a = \text{new address} \notin \Theta \end{array} \tag{A.35}$$

Evaluating the value of a key-value pair

$$\begin{array}{l} \langle \Theta, \gamma, S|\{d, \circ : e_v, \overline{\overline{e}}\}, a_k \rangle \\ \Rightarrow \langle \Theta, \gamma, S|\{d, a_k : \circ, \overline{\overline{e}}\}, e_v \rangle \end{array} \tag{A.36}$$

Evaluating key-value pairs in a dictionary literal

$$\begin{array}{l} \langle \Theta \quad , \gamma, S|\{d, a_k : \circ, \overline{\overline{e}}\} \quad , a_v \quad \rangle \\ \Rightarrow \begin{cases} \langle \Theta \quad , \gamma, S|\{d \oplus [(k \to a_v)], \circ : \overline{\overline{e}}_{1;2}, tail\ \overline{\overline{e}}\}, \overline{\overline{e}}_{1;1} \rangle & \text{if } |\overline{\overline{e}}| > 0 \\ \langle \Theta \oplus [a \to d \oplus [k \to a_v]], \gamma, S \quad , a \quad \rangle & \text{otherwise} \end{cases} \\ \textbf{where } a = \text{new address} \notin \Theta \\ \qquad k = \Theta(a_k) \end{array} \tag{A.37}$$

Dictionary primitives

The dictionary constructor primitive

$$\begin{array}{l} \langle \Theta \quad , \gamma, S, \texttt{dict.\_\_new\_\_}([a_c]) \rangle \\ \Rightarrow \langle \Theta \oplus [a \to ([\texttt{\_\_class\_\_} \to a_c], \emptyset)], \gamma, S, a \quad \rangle \\ \textbf{where } a = \text{new address} \notin \Theta \end{array} \tag{A.38}$$

The dictionary constructor primitive with starting dictionary

$$\begin{array}{l} \langle \Theta \quad , \gamma, S, \texttt{dict.\_\_new\_\_}([a_c, a_d]) \rangle \\ \Rightarrow \langle \Theta \oplus [a \to ([\texttt{\_\_class\_\_} \to a_c], \Theta(a_d))], \gamma, S, a \quad \rangle \\ \textbf{where } a = \text{new address} \notin \Theta \end{array} \tag{A.39}$$

The set item with key primitive

$$\begin{array}{l} \langle \Theta \quad , \gamma, S, \texttt{dict.\_\_setitem\_\_}([a_{\texttt{self}}, a_i, a_v]) \rangle \\ \Rightarrow \langle \Theta \oplus [a_{\texttt{self}} \to d'], \gamma, S, a_{\texttt{None}} \quad \rangle \\ \textbf{where } k = \Theta(a_i) \\ \qquad d' = (\Omega^{\Theta}_{a_{\texttt{self}}}, \Theta(a_{\texttt{self}}) \oplus [k \to a_v]) \end{array} \tag{A.40}$$

The get item with key primitive

$$\begin{array}{l} \quad \langle \Theta,\, \gamma,\, S,\, \texttt{dict.\_\_getitem\_\_}([\, a_{\texttt{self}},\, a_i\,]) \rangle \\ \Rightarrow \begin{cases} \langle \Theta,\, \gamma,\, S,\, \Theta(a_{\texttt{self}})(i) & \rangle \text{ if } i \in \Theta(a_{\texttt{self}}) \\ \langle \Theta,\, \gamma,\, S,\, \texttt{raise } a_{\texttt{IndexError}} & \rangle \text{ otherwise} \end{cases} \\ \textbf{where } i = \Theta(a_i) \end{array} \tag{A.41}$$

The delete item with key primitive

$$\begin{array}{l} \quad \langle \Theta \qquad\qquad\qquad\quad ,\, \gamma,\, S,\, \texttt{dict.\_\_delitem\_\_}([\, a_{\texttt{self}},\, a_i\,]) \rangle \\ \Rightarrow \langle \Theta \oplus [\, a_{\texttt{self}} \rightarrow d'\,],\, \gamma,\, S,\, a_{\texttt{None}} \qquad\qquad\quad \rangle \\ \textbf{where } k \;\; = \Theta(a_i) \\ \qquad\quad d' = (\Omega^{\Theta}_{a_{\texttt{self}}}, \Theta(a_{\texttt{self}}) \ominus k) \end{array} \tag{A.42}$$

Rules

6.1 Executing a non-empty block statement . . . 23
6.2 Executing an empty block . . . 23
6.3 Continuing execution of a block . . . 23
6.4 Pass statements . . . 24
6.5 Expression statements . . . 24
7.1 Executing an assignment statement . . . 25
7.2 Binding a variable . . . 25
7.3 Evaluating a variable expression . . . 26
7.4 Executing a variable deletion . . . 26
8.1 Executing a function definition statement . . . 28
8.2 Evaluating the function expression . . . 29
8.3 Evaluating the first argument expression . . . 29
8.4 Evaluating the remaining argument expressions . . . 29
8.5 Type checking function arguments . . . 30
8.6 Execution of a function . . . 30
8.7 Returning from the end of a function . . . 31
8.8 Executing a return statement . . . 32
8.9 Start stack unwinding . . . 32
8.10 Unwinding the stack . . . 32
8.11 Getting the next yielded value . . . 33
8.12 Getting the next value with yield value . . . 33
8.13 Evaluating a yield expression . . . 34
8.14 Start unwinding of the stack . . . 34
8.15 Unwinding the stack . . . 34
8.16 Stopping at the end of a generator . . . 34
9.1 Evaluating the bases of a new class . . . 38
9.2 Executing the body of a class definition . . . 38
9.3 Creating a new class . . . 38
9.4 The default class constructor . . . 39
9.5 Initializing the new class . . . 39
9.6 The default constructor . . . 40
9.7 The default object creation . . . 40
9.8 From creation to initialization . . . 40
9.9 The default object initialization . . . 40
9.10 Evaluating the attribute access expression . . . 41
9.11 Calling `__getattribute__` . . . 41
9.12 Getting attributes and instantiating function attributes . . . 41
9.13 Executing an instantiated function . . . 42
9.14 Getting class members and inserting type checks . . . 42
9.15 Evaluating the right hand side of an attribute assignment . . . 43

9.16 Evaluating the left hand side of an attribute assignment . . . 43
9.17 Calling `__setattr__` . . . 43
9.18 The default implementation of `__setattr__` for objects . . . 43
9.19 Evaluating the left hand side of an attribute deletion . . . 44
9.20 Calling `__delattr__` . . . 44
9.21 The default implementation of `__delattr__` . . . 44
10.1 Executing the raise statement . . . 45
10.2 Start unwinding of the stack . . . 46
10.3 Raising an exception . . . 46
10.4 Raising an exception in a function . . . 46
10.5 Raising an exception in a generator . . . 46
10.6 Executing the try-except statement . . . 47
10.7 Popping an exception clause while unwinding . . . 47
10.8 Catching an exception . . . 47
10.9 Popping the exception marker . . . 48
10.10 Executing the try-finally statement . . . 48
10.11 Popping a finally clause while unwinding . . . 48
10.12 Popping the finally marker . . . 49
11.1 Evaluating the test expression of an if statement . . . 51
11.2 Testing the test value of an if statement . . . 51
11.3 Executing the body of an if statement . . . 52
11.4 Evaluating the test expression of a while statement . . . 52
11.5 Testing the test value of a while statement . . . 53
11.6 Executing the body of a while statement . . . 53
11.7 Repeating a while statement . . . 53
11.8 Executing the for statement . . . 54
11.9 Get the iterator of the iterable . . . 54
11.10 Start the for loop with the iterator . . . 54
11.11 Starting an iteration of the for loop . . . 55
11.12 Executing the body of a for loop . . . 55
11.13 Stopping the for loop . . . 55
11.14 Rewinding the stack for a break statement . . . 55
11.15 Rewinding the stack for a continue statement . . . 56
12.1 Evaluating the printed expression . . . 57
12.2 Getting a string representation of the printed value . . . 57
12.3 Printing the string . . . 58
12.4 Faulty string representation . . . 58
13.1 Evaluating the operand of a unary operator expression . . . 59
13.2 Evaluating the unary not operator . . . 59
13.3 Return the negated value of a not operand . . . 60
13.4 Executing the function of a unary operator . . . 60
13.5 Evaluating the left hand side of a binary operator . . . 60
13.6 Evaluating the right hand side of a binary operator . . . 60
13.7 Evaluating an *is* expression . . . 61
13.8 Calling the operator attribute . . . 61
13.9 Lazy execution of the `and` operator . . . 61
13.10 Lazy execution of the `or` operator . . . 62

13.11 Evaluating the container expression . . . 62
13.12 Evaluating the key expression . . . 62
13.13 Calling `__getitem__` of a container . . . 63
13.14 Evaluating the right hand side expression . . . 63
13.15 Evaluating the container expression . . . 63
13.16 Evaluating the key expression . . . 63
13.17 Calling `__setitem__` . . . 64
13.18 Evaluating the container expression . . . 64
13.19 Evaluating the key expression . . . 64
13.20 Calling `__delitem__` . . . 64
14.1 Evaluating the string to be executed . . . 65
14.2 Evaluating the execution environment . . . 65
14.3 Execute the string . . . 65
14.4 Restoring the old environment . . . 66
14.5 Raising an exception in an execution environment . . . 66
14.6 Executing an import statement . . . 67
14.7 Importing modules or packages . . . 67
14.8 Returning from the import statement . . . 68
14.9 Raising an exception in an imported module . . . 68
A.1 The default string representation . . . 75
A.2 Default string representation for classes . . . 75
A.3 A function as a callable object . . . 76
A.4 The generator as an iterable . . . 76
A.5 Evaluating an integer literal . . . 77
A.6 The addition primitive . . . 77
A.7 The equals primitive . . . 77
A.8 The constructor primitive . . . 77
A.9 The string representation . . . 77
A.10 Evaluating a boolean literal . . . 78
A.11 The boolean to boolean conversion primitive . . . 78
A.12 The boolean to string conversion primitive . . . 78
A.13 Evaluating a string literal . . . 79
A.14 The string constructor primitive . . . 79
A.15 The string constructor primitive with starting string . . . 79
A.16 The length of string primitive . . . 79
A.17 The string addition operator primitive . . . 79
A.18 The string equals primitive . . . 80
A.19 The string get character at index primitive . . . 80
A.20 The string to string conversion primitive . . . 80
A.21 Evaluating a list literal . . . 80
A.22 Evaluating the elements of a list literal . . . 81
A.23 The list constructor primitive . . . 81
A.24 The length of list primitive . . . 81
A.25 The get item at index primitive . . . 81
A.26 The set item at index primitive . . . 81
A.27 The delete item at index primitive . . . 82
A.28 The append to list primitive . . . 82

A.29 The list addition operator primitive 82
A.30 Evaluating a tuple literal 82
A.31 Evaluating the elements of a tuple literal 83
A.32 The tuple constructor primitive 83
A.33 The length of tuple primitive 83
A.34 The get item at index primitive 83
A.35 Evaluating a dictionary literal 84
A.36 Evaluating the value of a key-value pair 84
A.37 Evaluating key-value pairs in a dictionary literal 84
A.38 The dictionary constructor primitive 84
A.39 The dictionary constructor primitive with starting dictionary 84
A.40 The set item with key primitive 84
A.41 The get item with key primitive 85
A.42 The delete item with key primitive 85

Bibliography

[1] D. Ancona, M. Ancona, A. Cuni, and N.D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. *Proceedings of the 2007 symposium on Dynamic languages*, pages 53–64, 2007.

[2] K. Barrett, B. Cassels, P. Haahr, D.A. Moon, K. Playford, and P.T. Withington. A monotonic superclass linearization for Dylan. *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 69–82, 1996.

[3] The CPython interpreter. `http://www.python.org`.

[4] New-style Classes. `ref:http://www.python.org/doc/newstyle/`.

[5] P. Hudak, J. Hughes, S.P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM Press New York, NY, USA, 2007.

[6] The IronPython interpreter. `http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython`.

[7] The Jython interpreter. `http://www.jython.org`.

[8] DE Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[9] Andres Löh. lhs2TEXa literal programming tool for Haskell. `http://people.cs.uu.nl/andres/lhs2tex/`.

[10] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19. *Computer Science Department, Aarhus University*, 1981.

[11] The PyPy interpreter. `http://pypy.org`.

[12] Michele Simionato. The Python 2.3 method resolution order. `http://www.python.org/download/releases/2.3/mro/`.

[13] G. van Rossum and F.L. Drake Jr. *Python Reference Manual*. Centrum voor Wiskunde en Informatica, 1995.

[14] Guido van Rossum. Overview of the Python enhancement proposals. `http://www.python.org/dev/peps/`.

[15] Guido van Rossum. Python Reference Manual. `http://docs.python.org/ref/`.