Semantics for CPython Virtual Machine

July 6, 2020

Variables & Scopes

Python has 4 kinds of variables.

The scope of Python is divided by function boundaries. No other Python language constructs create/destroy a scope.

Any name in a scope may denote 4 kinds of variables.

1. **local-only variable**
   A bound variable, but used only in the scope
2. **cell variable**
   A bound variable, the its definition scope has nested functions referencing this variable. This variable is in form of *cell* data structure, in case of future mutations
3. **free variable**
   A cell variable comes to a nested function referencing it, and becomes a free variable. It's also a *cell*
4. **global variable**
   Defined in the top-level of a Python module

These variables other than global variables are stored in variable *slots* created per function. When a function was created, an array of variables slots holding *local-only*, *cell* and *free* variables will be created as well. The index of a variable in the slots is statically decided by the bytecode compiler.

Particularly, **global variables** are stored in a special hash table, which could be accessed by a Python expression ***globals()***, and is separately maintained per Python module. Users can modify global variables outside its defined module, by modifying the special hash table.

CoreCPY: A Minimal Language for CPython Bytecode Instructions

A core part of Python VM instructions are given as follow

$$
\begin{aligned}
\langle \text{var} \rangle &::= \mathit{localvar} \mid \mathit{cellvar} \mid \mathit{freevar} \mid \mathit{globalvar} \\
\langle \text{instr} \rangle &::= \mathit{CONST}\ \mathit{literal} \\
\langle \text{instr} \rangle &::= \mathit{LOAD}\ \langle \text{var} \rangle \\
&\quad \mid \mathit{STORE}\ \langle \text{var} \rangle \\
&\quad \mid \mathit{JUMP\_IF\_TRUE}\ \mathit{int} \\
&\quad \mid \mathit{JUMP}\ \mathit{int} \\
&\quad \mid \mathit{CALL}\ \mathit{int} \\
&\quad \mid \mathit{POP}\ \mathit{int} \\
&\quad \mid \mathit{ROT}\ \mathit{int}
\end{aligned}
$$

Except for exception handling, the whole Python VM instruction set can be described in a set of primitive Python objects and this language.

We call this language Core CPython(**CoreCPY**), which is also we'd discuss in the following contents.

Extending this to support exception handling is not difficult, but it turns out to be annoying to do this, as the notations will loss conciseness and intuitions.

A translation from CPython 3.9 to CoreCPY is WIP.

To introduce the semantics of **CoreCPY**, we have to introduce Python function calls, a few built-in Python objects, in another word, *special objects*.

Python Function Calls

The protocol of Python function calls is complicated due to its dynamism for variadic arguments, keyword arguments and variadic keyword arguments.

```
# 'x, y, z' are positional or keyword
def f1(x, y, z): ...
f1(1, 2, 3)
f1(1, 2, z = 3)
f1(*(1, 2, 3))

# 'x, y, z' are positional-only
def f2(x, y, z, /): ...

# 'x' is positional or keyword, 'y, z' are keyword-only
def f3(x, *, y, z): ...

# 'variadic' is positional-variadic,
# 'f' accepts many positional arguments
def f4(*variadic): ...

def f5(**kw_variadic): ...
# etc..
...
```

Only one of its properties is relevant to the semantics of **CoreCPY**, which is that if we have another Python function object defined with a suite of instructions, when we call it via the **CALL** instruction, it should *at least* execute the instructions defining this function, if arguments are valid.

Due to the extreme complexity and inelegance, Python function calls and argument passing are so far treated **opaque** and **foreign** in this document.

As it's not very critical, we don't dig into its details.

Special Objects

Special objects are given as follow

$$
\begin{aligned}
\textbf{special\_object} = \ & \textbf{int} \\
| \ & \textbf{str} \\
| \ & \textbf{bool} \\
| \ & \textbf{True} \\
| \ & \textbf{False} \\
| \ & \textbf{None} \\
| \ & \textbf{type} \\
| \ & \textbf{getattr} \\
| \ & \textbf{setattr} \\
| \ & \textbf{globals} \\
| \ & \textbf{locals} \\
| \ & \textbf{make\_cell} \\
| \ & \textbf{make\_frozen}
\end{aligned} \tag{1}
$$

Beyond above special objects, all other Python objects can be implemented in Python itself or extensions written in C language. Only special objects concern Python's semantics.

All special objects other than **make_cell** and **make_frozen** are primitives in Python, and all special objects should support following functionalities, note that $\cdot(\cdot)$ means a Python function call

1. An **int** object should represent an integer with arbitrary precision
2. An object **s** typed **str** can support random accessing with any **int** index **i**, in form of ***getattr***(***str***, *"__getitem__"*)(*s*, *i*)
3. The **bool** object should satisfy ***type***(*True*) ***is bool*** and ***type***(*False*) ***is bool***
4. The **type** object is a function to access an object's type object
5. ***getattr***(*s*, *attr*) looks up object **s**'s attribute named **attr**; ***getattr***(*s*, *attr*, *default*) looks up object **s**'s attribute named **attr**, if not found, returns **default**
6. ***setattr***(*s*, *attr*, *value*) tries to set object **s**'s attribute named **attr** with **value**;
7. *make_cell()* makes a cell object, which is expected to be modified and read by ***setattr***(*cell*, *" cell_contents "*, *val*) and ***getattr***(*cell*, *" cell_contents "*)
8. *make_cell(o)* tells Python object **o** to reject any further modification to its attributes.
9. WIP

We also introduce some notations for abbreviations.

1. *a. attr* denotes ***getattr****(a, "attr").*
2. *a. attr* $<-$ *v* denotes ***setattr****(a, "attr", v); a. ;* here is an operator to compose two expressions, which means we firstly perform ***setattr****(a, " cell_contents ", v)*, and consequently return *a*. The same below.
3. *a["key"]* denotes ***getattr****(**type**(a), "__getitem__")(a, "key")*
4. *a["key"]* $<-$ *v* denotes ***getattr****(**type**(a), "__setitem__")(a, "key", v);**return** a*

.

Operational Semantics for CoreCPY

By introducing

- a value stack **S**,
- a store σ,
- an instruction suite **I**, and
- a function *const* to map a literal in **CoreCPY** language to a Python object

we get capable of clarifying the semantics of **CoreCPY**.

VAR

$$\frac{n \in \textbf{localvar}}{\sigma \vdash_{r-val} \Rightarrow \sigma(n)} \quad \frac{n \in \textbf{cellvar}}{\sigma \vdash_{r-val} \Rightarrow \sigma(n).cell\_contents}$$

$$\frac{n \in \textbf{freevar}}{\sigma \vdash_{r-val} \Rightarrow \sigma(n).cell\_contents} \quad \frac{n \in \textbf{globalvar}}{\sigma \vdash_{r-val} \Rightarrow globals()[n]}$$

$$\frac{n \in \textbf{localvar}}{\sigma \vdash_{l-var} \Rightarrow \lambda v.v} \quad \frac{n \in \textbf{cellvar}}{\sigma \vdash_{l-var} \Rightarrow \lambda v.\sigma(n).cell\_contents \leftarrow v}$$

$$\frac{n \in \textbf{freevar}}{\sigma \vdash_{l-var} \Rightarrow \lambda v.\sigma(n).cell\_contents \leftarrow v} \quad \frac{n \in \textbf{globalvar}}{\sigma \vdash_{l-var} \Rightarrow \lambda v.globals()[n] \leftarrow v}$$

INSTR

$$\frac{S = a_1 :: a_2 :: \cdots :: a_n :: S'}{S, \sigma \vdash_{instr} \textbf{POP}\ n \Rightarrow (S', \sigma)}$$

$$\frac{\sigma \vdash_{r-val} n \Rightarrow obj}{S, \sigma \vdash_{instr} \textbf{LOAD}\ n \Rightarrow (obj :: S, \sigma)}$$

$$\frac{}{S, \sigma \vdash_{instr} \textbf{CONST}\ l \Rightarrow (const(l) :: S, \sigma)}$$

$$\frac{\sigma \vdash_{l-val} n \Rightarrow L \quad S = obj :: S'}{S, \sigma \vdash_{instr} \textbf{STORE}\ n \Rightarrow (S', \sigma[n := L(obj)])}$$

$$\frac{S = a_1 :: a_2 :: \cdots :: a_n :: S'}{S, \sigma \vdash_{instr} \textbf{ROT}\ n \Rightarrow (a_n :: a_1 :: a_2 :: \cdots :: a_{n-1} :: S', \sigma)}$$

$$\frac{S = \textbf{locals} :: S'}{S, \sigma \vdash_{instr} \textbf{CALL}\ 0 \Rightarrow (\sigma :: S', \sigma)}$$

$$\frac{S = f :: a_1 :: a_2 :: \cdots :: a_n :: S' \quad f(a_1, a_2, \cdots, a_n) = e}{S, \sigma \vdash_{instr} \textbf{CALL}\ n \Rightarrow (e :: S', \sigma)}$$

JUMP

$$\frac{I(i) = \textbf{JUMP}\ off \quad I, S, \sigma \vdash_{jump} off => (return, \sigma')}{I, S, \sigma \vdash_{jump} i \Rightarrow (return, \sigma')}$$

$$\frac{I(i) = \textbf{JUMP\_IF\_TRUE}\ off \quad S = obj :: S' \quad type(obj).\_\_bool\_\_(obj) = True \quad I, S', \sigma \vdash_{jump} off => (return, \sigma')}{I, S, \sigma \vdash_{jump} i \Rightarrow (return, \sigma')}$$

$$\frac{I(i) = \textbf{JUMP\_IF\_TRUE}\ off \quad S = obj :: S' \quad type(obj).\_\_bool\_\_(obj) = False \quad I, S', \sigma \vdash_{jump} i + 1 => (return, \sigma')}{I, S, \sigma \vdash_{jump} i \Rightarrow (return, \sigma')}$$

$$\frac{I(i) = instr \quad instr \neq \textbf{JUMP}\ off_1 \wedge instr \neq \textbf{JUMP\_IF\_TRUE}\ off_2 \quad S, \sigma \vdash_{instr} \Rightarrow (S', \sigma') \quad I, S', \sigma' \vdash_{jump} i + 1 \Rightarrow (return, \sigma'')}{I, S, \sigma \vdash_{jump} \Rightarrow (return, \sigma'')}$$
