A Formalization of Python's Execution Machinery

Ammar Karkour[1] and Giselle Reis[1]

Carnegie Mellon University, Qatar
akarkour@andrew.cmu.edu, giselle@cmu.edu

Motivation. Python's increasing popularity has led to its adoption from entry-level programmers to scientists and engineers [1, 3, 4]. Hence, a trustworthy Python execution machinery should be critical and valuable. However, the language lacks a comprehensive formal definition that could be used to provide provable guarantees and guide a verified implementation [7, 8]. Previous attempts to formalize Python's source code have left out several features and core parts. This is in part due to the sheer size, complexity, and constant evolution of the language. But also, Python's definition is written in natural language (e.g. English), which can be imprecise, open to interpretation, and inconsistent with the actual implementation [7, 8].

Contribution. We propose that, since Python's virtual machine executes bytecode, an alternative direction towards a verified Python implementation is to start from this lower, smaller and more stable level. Therefore, we present, to our knowledge, the first formalization of Python's bytecode and virtual machine. This is, of course, not free of challenges, as Python's bytecode specification is also written in natural language. When descriptions were not clear, we used `cpython` as Python's reference implementation to fully understand the semantics. Our formalization uses inference rules in the style of [5, 6] to define typing of objects and semantics, which includes bytecode execution and frame stack management. The proposed rules are shown to satisfy progress and preservation. In addition, our proposed framework can be extended with built-in types without breaking safety guarantees. The formalized rules were implemented in F⋆, where properties can be proved automatically via dependent types or lemmas solved by an SMT solver. We call this implementation Py⋆ [1]. From the F⋆ implementation, we extracted a Python bytecode interpreter in OCaml. This verified Python execution machinery was compared to `cpython` for both consistency and performance.

Typing. Python is an object-oriented language in which all entities are objects of a certain class. However, unlike other object-oriented languages, there is no static class (or type) checking. Hence, one could say that all objects are of class `Object` statically, and their "real" class is only discovered at runtime. In that sense, Python is statically *unityped*, which means that "type checking" is now the responsibility of the execution machinery. Therefore, Python objects must have enough typing information so that the virtual machine is able to check types at runtime, and raise the appropriate errors when necessary. At the same time, this internal typing information should not impact how programmers see and operate with the objects. We achieve this by encapsulating the type information inside the top-level `Object` type.

Our typing system consists of 3 different layers `valTyp`, `cls`, and `pyObj`. At the innermost level is `valTyp`, indicating whether the class implements a built-in type (e.g. `int`, `string`, `list`, etc.) or is defined by the user (`USERDEF`). Below is a sample of the `valTyp` typing rules:

$$\frac{}{\texttt{USERDEF} : \texttt{valTyp}} \qquad \frac{i : \mathit{int}}{\texttt{INT}(i) : \texttt{valTyp}} \qquad \frac{s : \mathit{str}}{\texttt{STRING}(s) : \texttt{valTyp}} \qquad \frac{b : \mathit{bool}}{\texttt{BOOL}(b) : \texttt{valTyp}}$$

[1]The code for Py⋆ can be found here https://github.com/ammarkarkour/PyStar/

A `valTyp` value is encapsulated in a `cls` record, which is the type of all objects in Python's source code. This record contains the name of the class, the process id of the object, a `valTyp` value, and two mappings of fields and methods.

When it comes to execution, `cpython` uses the same type for both source code and virtual machine objects. E.g., a bytecode instruction and an integer would both have type `PyObject`. However, the distinction between these two kinds of objects is crucial for our formalization, as it allows proving correctness of the virtual machine independently of the user's code. This distinction is made via the constructors of `pyObj`, which are: `PYTYP`(obj) for `cls` objects; `CODEOBJECT`(co) for bytecode; `FRAMEOBJECT`(f) for frames (i.e. a program state); `FUN`(f) for functions on built-in Python types (e.g. < or `__lt__`); and `ERR`(s) for errors. Below is a sample of `pyObj` typing rules (in the interest of space, we will not show the typing rule for `frameObj`):

$$\frac{obj : \texttt{cls}}{\texttt{PYTYP}(obj) : \texttt{pyObj}} \qquad \frac{msg : str}{\texttt{ERR}(msg) : \texttt{pyObj}} \qquad \frac{f : list\ \texttt{pyObj} \rightarrow \texttt{valTyp}}{\texttt{FUN}(f) : \texttt{pyObj}} \qquad \frac{f : \texttt{frameObj}}{\texttt{FRAMEOBJECT}(f) : \texttt{pyObj}}$$

Semantics. Python's execution machinery works on a stack of *frames*. A *frame* is a tuple $\langle \varphi, \Gamma, i, \Delta \rangle$ where φ is a name context, Γ contains the bytecode Π, i is the program counter, and Δ is the data stack. The semantic rules formalize how frames f are evaluated and how the frame stack K is managed. A frame stack in the evaluation state is written as $K \vartriangleright f$, and in the return state as $K \vartriangleleft \mathsf{ret}(v)$. Frame stack evaluation rules use the judgment $K \circ f \longmapsto K' \circ f$, where $\circ \in \{\vartriangleright, \vartriangleleft\}$. Frame evaluation rules use the judgment $f \xmapsto{\Gamma.\Pi[i]} f'$, where the arrow is labelled with the bytecode operation being executed. An example of each kind of rule is shown below:

$$\frac{\langle \varphi, \Gamma, i, \Delta \rangle \xmapsto{\Gamma.\Pi[i]} \langle \varphi_n, \Gamma_n, i_n, \Delta_n \rangle}{K \vartriangleright \langle \varphi, \Gamma, i, \Delta \rangle \longmapsto K \vartriangleright \langle \varphi_n, \Gamma_n, i_n, \Delta_n \rangle} \qquad \frac{}{\langle \varphi, \Gamma, i, v :: \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\texttt{POP\_TOP}} \langle \varphi, \Gamma, i+1, \Delta \rangle}$$

Safety Proving safety (or soundness) of our typing system entails proving that well-typed terms do not reach a "stuck state", which is a state where no formal semantics rule is applicable [6]. This property is ensured by proving *progress* and *preservation* of our rules:

Thm 1 (Frame Stack Semantic Progress). *A well-typed frame stack does not get stuck, that is, it is either in a final state or it can take a step according to the frame stack semantic rules.*

Thm 2 (Preservation). *If an object $o : \tau$ evaluates to o', then $o' : \tau$.*

The proofs of the theorems follow the expected pattern. However, one must go through them to have at least a sanity check that all cases are covered.

Implementation. F⋆ is a general-purpose functional programming language with effects aimed at program verification [9]. One of the motivations for choosing F⋆ for our formalization was because the tool was successfully used to verify assembly instructions, which is a project close to ours [2]. Our verified implementation starts by embedding the defined types and objects in F⋆. Following that, we enforce the semantics rules' properties through the use of F⋆'s dependent types and lemmas. For example, this is how the rule for `POP_TOP` is implemented:

```
val pop_top: (l: list pyObj {Cons? l}) -> Tot (l2: list pyObj {l2 == tail l})
let pop_top datastack = List.Tot.Base.tail datastack
```

Following that, we use F⋆'s tools to extract a verified Python bytecode interpreter in OCaml, which was tested against hand-crafted test cases and a subset of `cpython`'s test kit. We are actively working on covering the whole of `cpython`'s test suite.

References

[1] Martin Desharnais and Stefan Brunthaler. Towards Efficient and Verified Virtual Machines for Dynamic Languages. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 61–75. Association for Computing Machinery, 2021.

[2] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A Verified, Efficient Embedding of a Verifiable Assembly Language. *Proc. ACM Program. Lang.*, 3(POPL), 2019.

[3] Samuel Groß. JITSploitation I: A JIT Bug. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html, 2020. Accessed: 2021-12-10.

[4] Samuel Groß. JITSploitation III: Subverting Control Flow. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-three.html, 2020. Accessed: 2021-12-10.

[5] Robert Harper. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press, 2016.

[6] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[7] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The Full Monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 217–232. Association for Computing Machinery, 2013.

[8] Python. Dis - disassembler for python bytecode. https://docs.python.org/3.8/library/dis.html#module-dis, 2022. Accessed: 2022-10-11.

[9] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.