Proceedings of the ESSLLI 2022 Student Session, 110–120
doi.org/10.21942/uva.20368209

Py⋆ : Formalization of Python's Verifiable Bytecode and Virtual Machine in F⋆

Ammar Karkour
akarkour@andrew.cmu.edu

Carnegie Mellon University, Qatar

Abstract. In order to avoid implementation bugs and inconsistencies of defining programming languages, computer scientists define formal semantics rules that guide the implementation process. In the case of Python, it lacks a formal implementation as it doesn't have formal semantics that describe the behavior of its complex functionalities. Previous attempts to provide formal implementation for Python didn't fully succeed. Since direct formalization of Python source code is hard, in this project, we define formal semantics rules for Python's Bytecode instead, and embed them in the theorem prover F⋆. Following that we extract efficient executable OCaml code of our embedding, which could be used to interpret Python Bytecode and to find bugs in other interpreters.

Keywords: Formal Semantics · Programming Languages · Python · F⋆

1 Introduction

When writing a program, it is very important that the written code matches its specifications. By doing this, we have the potential of eliminating software bugs that cost around $1.56 trillion yearly [1]. Similarly, it is hard to trust any program written in a language that does not match its own specifications. To address this problem, computer scientists formalize programming languages using formal semantics rules that describe how each expression in the language behave. These guide the implementation process through eliminating ambiguities.

The comparisons between different C compilers such as LLVM, GCC, and CompCert provide strong evidence of the power of formalization and verification to reduce implementation errors [11]. Having formally verified compilers guarantees a trustworthy execution machinery, and helps in finding bugs in non-verified compilers and interpreters. For example, CompCert reported 325 new bugs to compiler developers through comparing the results of running correct randomly generated C code on CompCert against other commonly used C compilers [11].

Python is one of the most used programming languages across industry, science, education, and many other fields. This heavy dependence on the language makes a trustworthy execution machinery very critical and highly valuable. However, a closer look at Python's virtual machines shows that this confidence is unwarranted, as through abuse of implementation errors, faults could be exploited and attackers can hijack victim devices through arbitrary code execution [4,6,7].

Even though Python has become one of the most important languages, it still lacks a formal implementation and certain formal verification methods because it was not designed with formal rigor. This is the reason that Python still lacks formal semantics that formalizes and describes the behavior of its complex functionalities. Instead, Python's semantics is described in a documentation written in English (natural language), which suffers from several problems compared to formal semantics. The first problem is that written documentation could be interpreted differently depending on who is reading it. Secondly, they are not precise nor accurate, which makes keeping track of all different states and cases nearly impossible.

Examples of the problems of Python's documentation include: how the documentation of the Bytecode instruction `LOAD_FAST` is not consistent with the implementation, how the documentation of the compare operations is vague, and how the documentation of `__eq__()` is not precise and can easily deceive a programmer into making hard to find bugs. Check Appendix A for more details about these examples.

Formalizing Python source code is hard because of its many complex functionalities and inconsistencies, so a more realistic and attainable goal is to formalize Python's Bytecode. This way we focus on a smaller set of simpler instructions, that are easier to formalize and formally verify. Moreover, in the end, Bytecode is what is being executed by the interpreter. Therefore, the goal of the project is: **implementing an efficient verified embedding of Python's virtual machine and verifiable Python's Bytecode in F⋆. We call this implementation Py⋆** .

F⋆ (pronounced F star) is a general-purpose functional programming language with effects aimed at program verification. It puts together the automation of an SMT-backed deductive verification tool with the expressive power of a proof assistant based on dependent types. After verification, F⋆ programs can be extracted to efficient OCaml, F#, C, WASM, or ASM code. This enables verifying the functional correctness and security of realistic applications [3]. F⋆ has been successfully used for multiple formalization and verification projects in the field of programming languages and software security. One of these projects is the embedding of verifiable Assembly in F⋆, which has a very similar goal to our project Py⋆ [5].

The contribution of this paper conatins the development of three parts: (1) defining formal semantics rules for Python's Bytecode, (2) embedding Python's Bytecode and implementing the virtual machine in F⋆, then proving that our implementation matches the defined formal semantics using F⋆'s automated theorem prover Z3, (3) extracting an efficient executable code of the formalized virtual machine in OCaml using F⋆ code extraction tool.

Py⋆ code can be found here.[1]

[1] https://github.com/ammarkarkour/PyStar

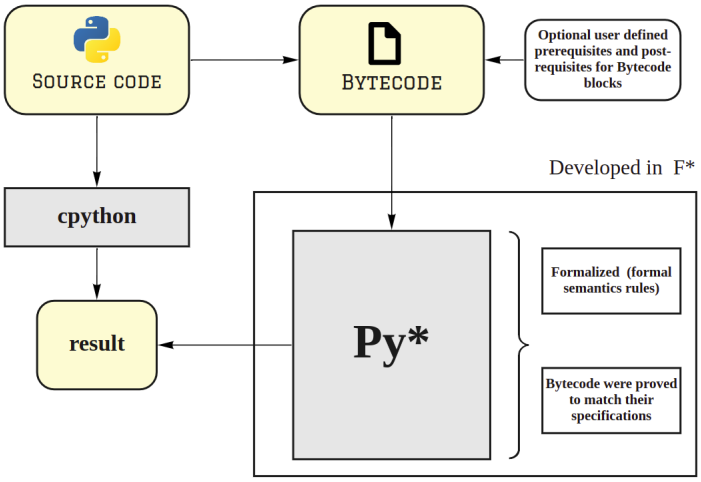

Fig. 1. Overview of Py⋆

2 Related Work

A previous attempt to formalize Python was made in [10], where the authors introduced small-step semantics rules for Python and implemented a subset of these semantics as a core language. They also present a translation process from Python source code to their core language. However, that core language is not exactly Python, but rather a different language that Python code needs to be translated to before running it. Moreover, several features of Python and the defined semantics were left unimplemented or unfinished, such as special fields and complex scope cases. Lastly, the resulting interpreter suffered a performance hit, because the translation process to the target language was too costly. For instance, test cases that usually take a few seconds to be executed with cpython, took around 20 minutes [10]. Since Py⋆ does not have the translation process, it will be interesting to compare its performance with the performance of their formalization.

In his PhD thesis [8], Monat defined concrete semantics of a large subset of Python source code, with the goal of statically analyzing Python code to avoid software faults. In his thesis he describes the implementation of the abstracted rules in Mopsa. However, even though the provided concrete formal semantics cover most of Python's features, it is not executable in itself. Instead, they test it through Python's value analysis in Mopsa, which is a close implementation that perform over-approximations. Implementing an executable version of the defined concrete semantics and proving its correctness is left as future work.

The clear difficulty of direct formalization of languages like Python resulted in a goal switch: from formalizing the whole language, to formalizing its execution model. In 2021, Desharnais and Brunthaler presented a system where

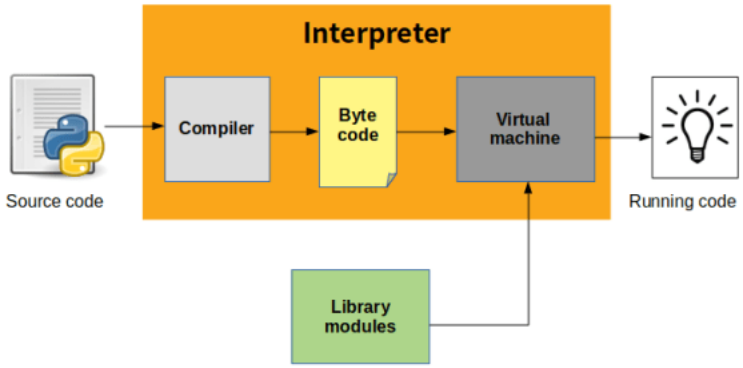

Fig. 2. Python Code Execution Mechanism

they defined an alternative self-optimizing Bytecode interpreter for dynamically typed programming languages in the proof assistant Isabelle/HOL [4]. Their interpreter consists of three Bytecode languages and their virtual machines that support Just-In-Time (Jit) compilation. Even though the default implementation of Python (cpython) does not support Jit compilation, they mentioned that the overall idea and approach used in implementing their interpreter could be used for implementing a verified Python virtual machine.

3 Formal Semantics

Even though Python doesn't have formal semantics, it has a default implementation – **cpython**. We focus on cpython version 3.9 (released in October 2020). Through analyzing the source code of Bytecode evaluation in cpython and its documentation, we understood the behavior of Python and formalized it using **small-step semantic** rules.

To be executed, Python source code is first translated into a **code object**. The virtual machine holds the code object to be executed and a **call stack**. The call stack manages **frames**. Frames are created and deleted during execution, and they are used to provide context for Bytecode blocks (see Figure 3). In other words, a frame acts as the program state of a Bytecode block.

Once the virtual machine is created it spawns the initial frame to run the Bytecode of the code object in it. The virtual machine starts the execution process which involves spawning new frames, executing them and then returning the results to caller frames. Once the initial frame returns a result, the execution process ends and the interpreter returns the result (see Figure 2).

Our small-step semantics consist of two levels (i.e. two types of rules), **Frame Instructions**, and **Bytecode Instructions**. The former describes how frames

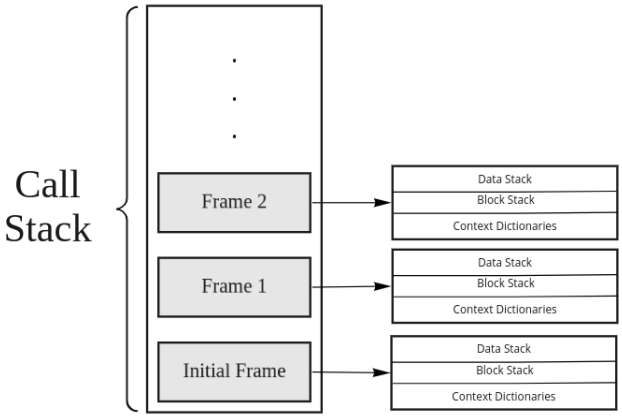

Fig. 3. Python Virtual Machine

are managed and how they interact with each other (i.e. data flow between frames). The latter describes the execution steps for each Bytecode instruction (i.e. execution within the frames).

3.1 Notation

In this section we introduce the notation used in the small-step semantics. We assume that equality is commutative in all rules.

The *call stack* has two states: an *evaluation state* $\mathsf{K} \triangleright \mathsf{f}$, and a *return state* $\mathsf{K} \triangleleft \mathsf{ret}(\mathsf{v})$, where K is a stack of frames, f is a frame, and v is a value. During the evaluation state we evaluate the frame f until it becomes $\mathsf{ret}(\mathsf{v})$, switching to the return state. The empty stack is denoted ϵ.

Frames are represented by a tuple $\langle \varphi, \Gamma, i, \beta, \Delta \rangle$, where:

$$
\begin{aligned}
\varphi &\triangleq \langle \Sigma_{\mathsf{g}}, \Sigma_{\mathsf{l}}, \Sigma_{\mathsf{l}_+} \rangle && : \text{Contexts} \triangleq \langle \text{ global names, local names, local+ } [2] \rangle \\
\Gamma &\triangleq \langle \Pi, \Sigma_{\mathsf{c}}, \Sigma_{\mathsf{v}}, \Sigma_{\mathsf{n}} \rangle && : \text{Code Object} \triangleq \langle \text{ Bytecode, constants, varnames, names } \rangle \\
i & && : \text{Program Counter} \\
\beta &\triangleq \langle \mathsf{t}, \mathsf{l}, \mathsf{h} \rangle && : \text{Block Object} \triangleq \langle \text{ type, level, handler } \rangle \\
\Delta & && : \text{Data Stack}
\end{aligned}
$$

Frame level semantics use $\longmapsto$ to indicate *stepping*, while Bytecode instructions level use $\xmapsto{\Gamma.\Pi[i]}$ to indicate stepping.

Python is an object-oriented language where all entities in it are objects. Objects in Python belong to either a built-in class or a user-defined class. Two things are needed to create an object, the class of the object, and the value of

[2] This field enables the evaluation loop to optimize loading and storing values of names to and from the value stack with the `LOAD_FAST` and `STORE_FAST` instructions.

the object. Both of these two things form the type of the object. For example, the number `5` in Python is an object that belongs to the class `INT` and has a value of 5, so its type is `INT(5)`. To formalize this relation, we use the following auxiliary terms:

T	: Builtin classes: {`INT`, `BOOL`, `STRING`, `LIST`, `TUPLE`, `DICT`, `FUNCTION`, `NONE`}
`USERDEF`	: User-defined class
`C`(val)	: Type constructor, where `C` $\in T$ or `C` = `USERDEF`, and val is value
`CreateObj`(t)	: Object constructor: takes in `C`(val) and constructs an object
`TypeOf`(obj)	: Object destructor $\triangleq$ Takes an object as an input and returns `C`(val)

3.2 Frame Instructions

Frame Instructions describe how frames are managed and how they interact with each other (i.e. data flow between frames). To demonstrate how these rules work, we explain Rule 4 as an example. In Rule 4, the call stack is in a return state where the evaluation of frame $\langle \varphi, \Gamma, i, \beta, v :: \Delta \rangle$ just finished, and it must return to its caller stack. The premise of the Rule ensures that the returned value is not a frame (that would need to be spawned). If that is the case, the top element v on the data stack is pushed to the data stack of the caller frame, the returned frame is discarded, and execution continues on an evaluation state of the caller frame. In addition, Rule 1 denotes function execution, Rule 2 denotes a function return, and Rule 3 a function call.

Frame Rules

$$\frac{\langle \varphi, \Gamma, i, \beta, \Delta \rangle \xrightarrow{\Gamma.\Pi[i]} \langle \varphi_n, \Gamma_n, i_n, \beta_n, \Delta_n \rangle}{K \ \triangleright \ \langle \varphi, \Gamma, i, \beta, \Delta \rangle \longmapsto K \ \triangleright \ \langle \varphi_n, \Gamma_n, i_n, \beta_n, \Delta_n \rangle} \quad (1)$$

$$\frac{\langle \varphi, \Gamma, i, \beta, \Delta \rangle \xrightarrow{\Gamma.\Pi[i]} \mathsf{ret}(\langle \varphi_n, \Gamma_n, i_n, \beta_n, \Delta_n \rangle)}{K \ \triangleright \ \langle \varphi, \Gamma, i, \beta, \Delta \rangle \longmapsto K \ \triangleleft \ \mathsf{ret}(\langle \varphi_n, \Gamma_n, i_n, \beta_n, \Delta_n \rangle)} \quad (2)$$

$$K \ \triangleleft \ \mathsf{ret}(\langle \varphi, \Gamma, i, \beta, \langle \varphi_n, \Gamma_n, i_n, \beta_n, \Delta_n \rangle :: \Delta \rangle) \longmapsto K; \langle \varphi, \Gamma, i+1, \beta, \Delta \rangle \ \triangleright \ \langle \varphi_n, \Gamma_n, i_n, \beta_n, \Delta_n \rangle \quad (3)$$

$$\frac{v \neq \texttt{FRAMEOBJECT}(fo)}{K; \langle \varphi_p, \Gamma_p, i_p, \beta_p, \Delta_p \rangle \ \triangleleft \ \mathsf{ret}(\langle \varphi, \Gamma, i, \beta, v :: \Delta \rangle) \longmapsto K \ \triangleright \ \langle \varphi_p, \Gamma_p, i_p, \beta_p, v :: \Delta_p \rangle} \quad (4)$$

3.3 Bytecode Instructions

As the number of rules for this section is big, we only include an example of rules in each class. The rest of the rules are either included in this *file* [3], or they follow the same structure as the mentioned rules so they are not included.

Bytecode Instructions describe the execution steps for each Bytecode instruction (i.e. execution within the frames). They are divided into mainly 4 categories (General, Unary, Binary, and Miscellaneous) based on their functionalities as the

[3] https://github.com/ammarkarkour/PyStar/blob/main/formal_semantics.pdf

rules show. To demonstrate how these rules work, we explain how Rule 9 works as an example. Rule 9 states that if the current Bytecode instruction that the program counter `i` pointing to is `BUILD_LIST(n)`, and the data stack contains at least n elements, then we pop these n elements and we create a list that contains them $\texttt{LIST}([v_1, ..., v_n])$. Following that, we push the created list back to the data stack, and we increment the program counter by 1.

Similarly, Rule 5 denotes raising an error, Rule 6 denotes removing the top element in the data stack, Rule 7 denotes Python's logical `not`, Rule 8 denotes Python's floor division (i.e., `//`), and Rule 10 denotes the Bytecode instruction for functions calls.

General Instructions

$$\frac{}{\langle \varphi, \Gamma, i, \beta, \texttt{ERR}(s) :: \Delta\rangle \xrightarrow{\Gamma.\Pi[i]} \mathsf{ret}(\langle \varphi, \Gamma, i, \beta, \texttt{ERR}(s) :: \Delta\rangle)} \quad (5)$$

$$\frac{}{\langle \varphi, \Gamma, i, \beta, v :: \Delta\rangle \xrightarrow{\Gamma.\Pi[i]=\texttt{POP\_TOP}} \langle \varphi, \Gamma, i+1, \beta, \Delta\rangle} \quad (6)$$

Unary Instructions

$$\frac{\texttt{not v} = \texttt{C}(v') \quad v'' = \texttt{CreateObj}(\texttt{C}(v'))}{\langle \varphi, \Gamma, i, \beta, v :: \Delta\rangle \xrightarrow{\Gamma.\Pi[i]=\texttt{UNARY\_NOT}} \langle \varphi, \Gamma, i+1, \beta, v'' :: \Delta\rangle} \quad (7)$$

$$\frac{v = \texttt{INT}(\bar{v}) \quad \bar{v} = 0}{\texttt{not } v = \texttt{BOOL}(true)} \qquad \frac{v = \texttt{STRING}(\bar{s}) \quad \bar{s} \neq \texttt{""}}{\texttt{not } v = \texttt{BOOL}(false)} \qquad \frac{v = \texttt{BOOL}(true)}{\texttt{not } v = \texttt{BOOL}(false)}$$

Binary Instructions

$$\frac{\texttt{TypeOf}(v1) = \texttt{C}(\_) \quad \texttt{C} \in T \quad \texttt{TypeOf}(v2) = \texttt{C'}(\_) \quad \texttt{C'} \in T \quad v2/v1 = \texttt{C''}(v) \quad v' = \texttt{CreateObj}(\texttt{C''}(v))}{\langle \varphi, \Gamma, i, \beta, v1 :: v2 :: \Delta\rangle \xrightarrow{\Gamma.\Pi[i]=\texttt{BINARY\_FLOOR\_DIVIDE}} \langle \varphi, \Gamma, i+1, \beta, v' :: \Delta\rangle} \quad (8)$$

$$\frac{v1 = \texttt{INT}(\bar{v1}) \quad v2 = \texttt{INT}(\bar{v2}) \quad \bar{v1} \neq 0 \quad v' = \bar{v2}/\bar{v1}}{v2/v1 = \texttt{INT}(v')}$$

$$\frac{v1 = \texttt{INT}(\bar{v1}) \quad v2 = \texttt{INT}(\bar{v2}) \quad \bar{v1} = 0}{v2/v1 = \texttt{ERR}(s)}$$

Miscellaneous opcodes

$$\frac{l = \texttt{LIST}([v_1, ..., v_n])}{\langle \varphi, \Gamma, i, \beta, v_1 :: ... :: v_n :: \Delta\rangle \xrightarrow{\Gamma.\Pi[i]=\texttt{BUILD\_LIST(n)}} \langle \varphi, \Gamma, i+1, \beta, l :: \Delta\rangle} \quad (9)$$

$$\frac{v = \texttt{CODEOBJECT}(\bar{co}) \quad f = \langle\langle \varphi.\Sigma_g, \{\}, [v_n, \ \ldots, \ v_1]\rangle, \bar{co}, 0, [], []\rangle}{\langle \varphi, \Gamma, i, \beta, v_1 :: ... :: v_n :: v :: \Delta\rangle \xrightarrow{\Gamma.\Pi[i]=\texttt{CALL\_FUNCTION(n)}} \mathsf{ret}(\langle \varphi, \Gamma, i, \beta, f :: \Delta\rangle)} \quad (10)$$

4 Implementation

The first step of implementing the virtual machine according the formal semantics defined in Section 3 is embedding the types and objects in F⋆. To do that we use F⋆'s typing system to represent the different components of the virtual machine. In this section we discuss the embedded types and objects that are currently supported by Py⋆ (see Appendix A for the full embedding).

Bytecode is encoded as values of type `bytecode` constructed as `CODE l`. Each bytecode instruction has type `opcode` and `l` has type `list opcode`.

```
type opcode = | NOP: opcode
              | POP_TOP: opcode
              | ROT_TWO: opcode
              | ROT_THREE: opcode
                ...
type bytecode = | CODE: l: list opcode -> bytecode
```

The virtual machine contains different stacks that manage the values needed during evaluation. All values stored in these stacks have type `pyObj`. `pyObj` has 3 constructors: `PYTYP`(obj), `CODEOBJECT`(co), and `FRAMEOBJECT`(fo).

Python classes are records of type `cls` which includes a `name`; a process ID (`pid`); a `value` (used for builtin classes, e.g., int); and `fields` and `methods` which store the class' fields and methods. We use F⋆'s `Map` module for mapping names of fields and methods to their values. Following the literature [9], we represent classes as *records*.

```
type cls = { name: string;
             pid: int;
             value: builtins;
             fields: Map.t string pyObj;
             methods: Map.t string pyObj }
```

PYTYP(obj) is the type of every object in a Python program, where obj is of type `cls`, be it from a user-defined class or a built-in type. Classes and objects are formalized in F⋆ as types and values of these types, respectively.

CODEOBJECT(co) is the input for the virtual machine. It is represented by a record with four fields: `co_code` contains the Bytecode that will be executed in the initial frame; `co_consts` contains constants like string literals, numeric values, and other code objects that are needed for execution; `co_varnames` contains locally defined names in a code block; finally `co_names` contains a collection of non-local names used within the code object.

```
type codeObj = { co_code: bytecode;
                 co_consts: list pyObj;
                 co_varnames: list string;
                 co_names: list string; }
```

FRAMEOBJECT(fo) As shown in section 3, a frame acts as a program state for a Bytecode block. The representation comes directly from the rules explained in section 3.1. Records are also used to represent a frame, as shown below.

```
type frameObj = { dataStack: list pyObj;
                  blockStack: list blockObj;
                  fCode: codeObj;
                  pc: nat;
                  f_localplus: list pyObj;
                  f_globals: Map.t string pyObj;
                  f_locals: Map.t string pyObj; }
```

4.1 Execution and Formal Verification

Py⋆ runs a frame by traversing its Bytecode instructions using the program counter. Each instruction is executed by a helper function that takes as input only what is needed for that instruction. It returns the updated components after the instruction is executed.

Through its syntax and semantics, formal semantics rules contain the properties that state the correct behavior of the Bytecode instructions. To ensure that our implementation has the properties that our rules state, we use F⋆'s dependent types and automatic theorem prover (Z3) to enforce the properties at type checking. More specifically, we deduce pre-conditions and post-conditions from the formal semantics rules, and we ensure that the helper functions that correspond to these rules respect them through their types.

For example, the type of the helper function that implements the instruction `POP_TOP` ensures that given a datastack that is not empty, it removes the top element in the stack and returns the rest of the stack as described in rule 6. Moreover, since the function is `Total`, no unexpected behavior will happen.

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

5 Results and Conclusion

As far as we know, Py⋆ is the first formalization of Python's bytecode. Using the generated interpreter, Python programmers can safely run their code without worrying about unexpected behavior. Computer scientists would find in our formal semantics a precise explanation of the behavior of Python's interpreter, which could be used to guide the implementation of new Python interpreters. It also could be used to find new bugs in current commonly used interpreters similar to how CompCert did with different C compilers through an automated testing pipeline. Lastly, since we have a function in F⋆ that runs Bytecode blocks, one can use F⋆'s theorem proving modules to prove properties about such blocks.

Py⋆ currently supports a verified implementation of instructions that allow us to interpret Python programs that contain ints, strings, bools, lists, dictionaries, tuples, none, classes, exceptions, variable assignment, function calls, loops, conditionals, and context switching. These features were also further tested using a group of hand crafted test cases.

As future work, we are working on mainly three things: (1) supporting the rest of Python's features such as Floats and Sets by writing formal semantics rules for them, implementing them, and proving that the implementation matches the formal semantics; (2) more extensive testing for our system will be done through cpython's test kit that is full of different test cases that could be used for correctness and performance checks; (3) implementing an automated testing tool (similar to CompCert's tool) that could be used to find bugs in other Python interpreters [4]. Lastly, it would be interesting to try proving properties about different Bytecode blocks using F⋆'s proving modules.

References

1. https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf, accessed: 2022-3-21
2. dis — disassembler for python bytecode — python 3.10.3 documentation. https://docs.python.org/3/library/dis.html, accessed: 2022-3-21
3. F*: A Higher-Order Effectful Language Designed for Program Verification. https://www.fstar-lang.org/, accessed: 2021-12-10
4. Desharnais, M., Brunthaler, S.: Towards Efficient and Verified Virtual Machines for Dynamic Languages. In: Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 61–75. CPP 2021, Association for Computing Machinery (2021). https://doi.org/10.1145/3437992.3439923
5. Fromherz, A., Giannarakis, N., Hawblitzel, C., Parno, B., Rastogi, A., Swamy, N.: A Verified, Efficient Embedding of a Verifiable Assembly Language. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290376
6. Groß, S.: JITSploitation I: A JIT Bug. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html (2020), accessed: 2021-12-10
7. Groß, S.: JITSploitation III: Subverting Control Flow. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-three.html (2020), accessed: 2021-12-10
8. Monat, R.: Static type and value analysis by abstract interpretation of Python programs with native C libraries. Ph.D. thesis, Sorbonne Université (2021)
9. Pierce, B.C.: Types and programming languages. MIT press (2002)
10. Politz, J.G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., Krishnamurthi, S.: Python: The Full Monty. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. p. 217–232. OOPSLA '13, Association for Computing Machinery (2013). https://doi.org/10.1145/2509136.2509536
11. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and Understanding Bugs in C Compilers. SIGPLAN Not. **46**(6), 283–294 (2011). https://doi.org/10.1145/1993316.1993532

[4] Python interpreters that use cpython's Bytecode, since not all interpreters use it.

A Issues in Python Documentation

In this Appendix we show some examples of the inconsistencies and ambiguities that Python documentation contains.

- `LOAD_FAST var_num`
  When executing Python source code, it first gets compiled into Bytecode instructions, and then these instructions get executed by the interpreter (i.e. Virtual Machine). `LOAD_FAST` is one of these instructions. Python's documentation reads "`LOAD_FAST var_num`: Pushes a reference to the local `co_varnames[var_num]` onto the stack" [2]. However, by taking a look at Python's default implementation (**cpython**), you can clearly see that `co_varnames` only stores the names not the values associated with these names, and what really happens is that they load `f_localplus[var_num]` to the stack. This is just one example of many where the description of a Python instruction in the documentation is not consistent with the implementation, which could lead to different behaviors from different interpreters.

- **compare operations**
  The documentation is also full of ambiguous descriptions. For example, they state that "The `<, <=, >` and `>=` operators are only defined where they make sense; for example, they raise a `TypeError` exception when one of the arguments is a complex number" [2]. Such a sentence is very vague as they do not define what *makes sense* means in this situation. These ambiguous descriptions cannot be used to define something formal, and they form a big danger as a source of potential implementation bugs.

- `__eq__()`
  Another example of how how the documentation can deceive a programmer into making hard to find bugs is the documentation of `__eq__()`. The documentation states that "Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method" [2], which indicates that if you don't define the `__eq__()` method of a specific class then `a.__eq__(b)` where a and b are objects of that class returns `False`. However, in practice `a.__eq__(b)` returns `NotImplemented`, where `NotImplemented` is a special value (not an error) that is usually returned in such situations. Moreover `NotImplemented` has a Boolean value of `True`, so `a.__eq__(b)` always returns `True`. For example, the code shown below would actually print `SHOULD NOT HAPPEN`.

```
class Car():
    pass

a = Car()
b = Car()

if a.__eq__(b):
    print("SHOULD NOT HAPPEN")
```