# Py*: A Formalization of Python's Execution Machinery

Ammar Karkour and Giselle Reis
Carnegie Mellon University

**Types Conference**
**Mon 13 June 2023**

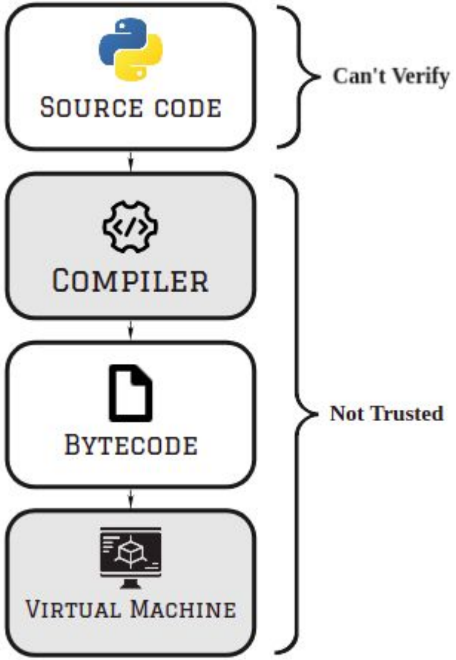
# Overview

- Motivation
- Research Contribution
- Typing Rules
- Evaluation Rules
- Discussion and Future Work
- Q & A

# Motivation

# Challenges of Formalizing Python

- Python was not designed with formal rigor.
- The language extends and grows very fast.
- **Formality**: Python source code doesn't have formal semantics.
- **Extendability**: It is hard to keep track with all different components.



# Example of Python's Informality

Documentation:

"Non-identical instances of a class normally compare as non-equal unless the class defines the __eq__() method or the __cmp__() method"

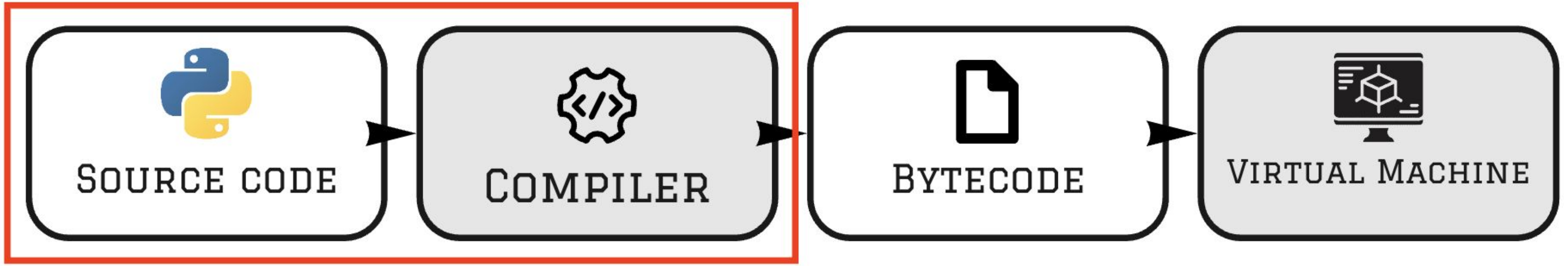
Reality:

```
class Car():
    pass

a = Car()
b = Car()

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

Reason:

- a.__eq__(b) returns NotImplemented
- **NotImplemented** has a Boolean value of **True**

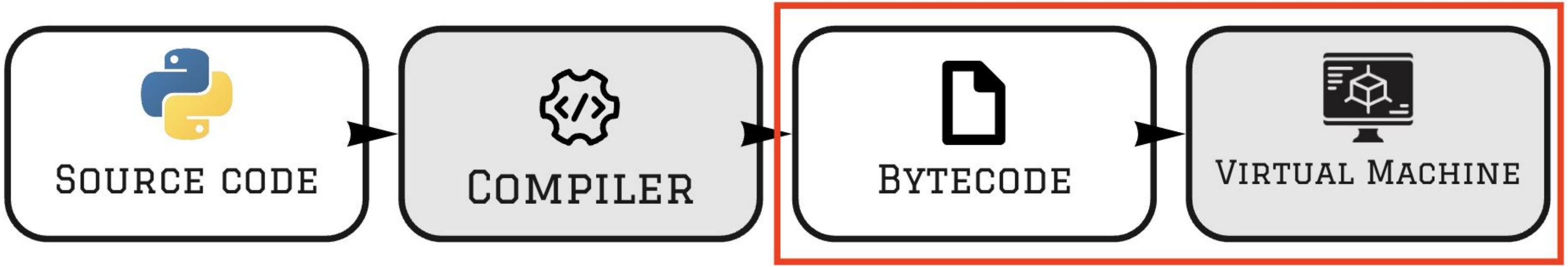
# Previous Formalization Attempts




- Previous work focus on formalizing Python's Source code.
    - Forces them to handle very high-level concepts that hide a lot of complexities underneath.

# Research Contribution

# Focus on Bytecode




- Why?
  - **Simpler** and **smaller** set of instructions.
  - More stable than Source code which implies **easier extendability.**
  - **It's  what gets executed at the end.**

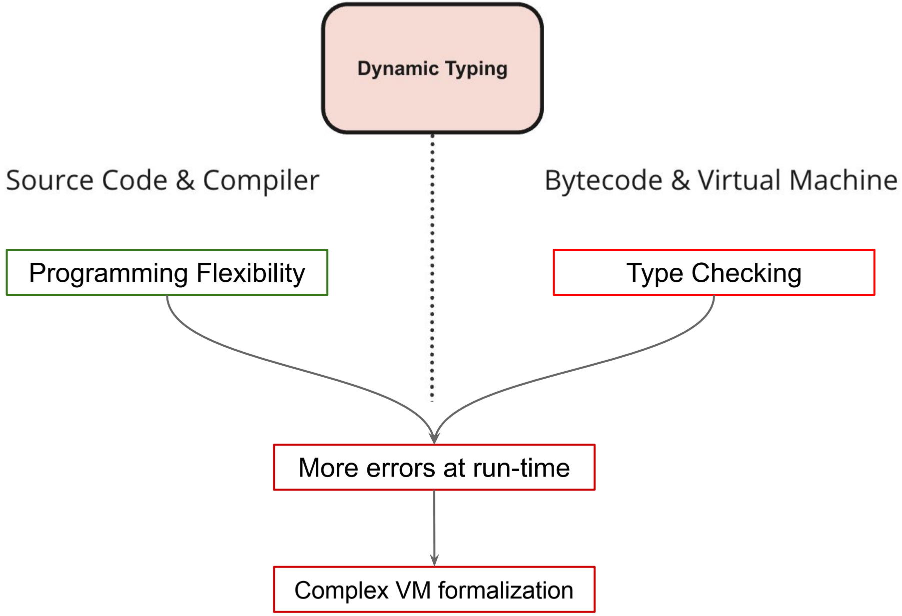
# Contribution

- **Bytecode formal semantics**
  - Formal semantics for typing and evaluation rules.
  - The system can be **extended** with rules for built-in classes.
  - **Safety** by proving progress and preservation.
- **Py***
  - A formally verified implementation of the rules in **F***.
- **Formally Verified Python Virtual Machine**
  - Finally we extract a formally verified executable OCaml code of Py*.

# Typing

# Challenges

- Python is a dynamically typed object-oriented language:
    - All entities in a python program have the same type called ***object***.
    - At the source code level Python is ***Statically uni-typed***.



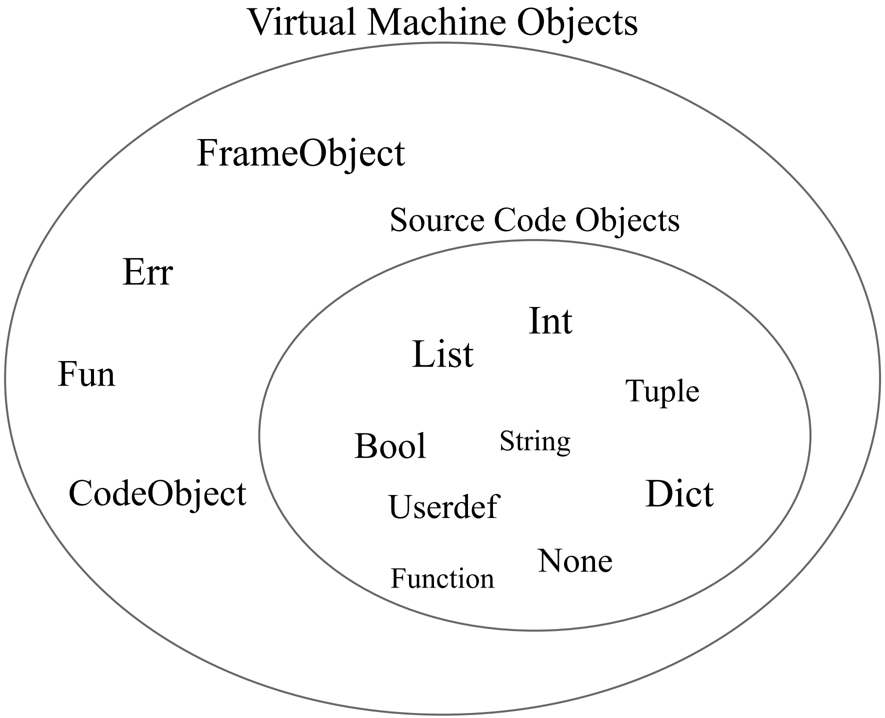
# Goals

*To maintain flexibility and ease of programming in Python while having the safety guarantees that we usually have with Static typing.*

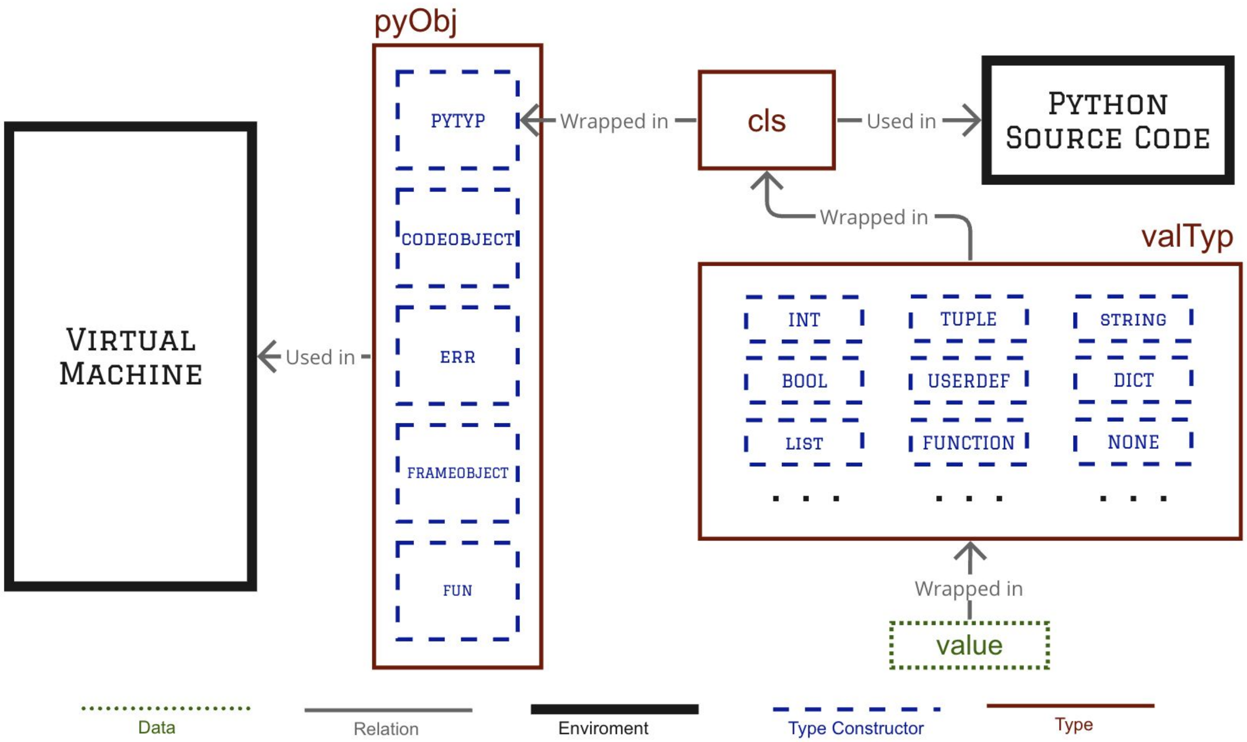
*To ensure practicality by having a modular and extendable design for the typing system.*

# Program Entities

- To ensure modularity and safety:
  - Separate Source code Objects away from VM objects.
  - Define an interface that formalizes the interactions between Source Code Objects and VM Objects.
  - Entails:
    - If new objects are added then nothing that was already built breaks.



# Typing System Design



# Definitions of Program States

- Py* Functions on Frames which behaves as a program states.

*Definition 3.1 (Frame).* A *frame* is a tuple $\langle \varphi, \Gamma, i, \Delta \rangle$ where:

$$\begin{aligned}
\varphi \triangleq \langle \Sigma_g, \Sigma_l, \Sigma_{l_+} \rangle \quad &: \quad \text{Contexts} \triangleq \langle \text{ global names, local names, local+}^1 \rangle \\
\Gamma \triangleq \langle \Pi, \Sigma_c, \Sigma_v, \Sigma_n \rangle \quad &: \quad \text{Code Object} \triangleq \langle \text{ bytecode, constants, varnames, names } \rangle \\
i \quad &: \quad \text{Program Counter} \\
\Delta \quad &: \quad \text{Data Stack}
\end{aligned}$$

$$\frac{obj : \texttt{cls}}{\texttt{PYTYP}(obj) : \texttt{pyObj}} \qquad \frac{co : \texttt{codeObj}}{\texttt{CODEOBJECT}(co) : \texttt{pyObj}} \qquad \frac{msg : str}{\texttt{ERR}(msg) : \texttt{pyObj}}$$

$$\frac{f : \texttt{frameObj}}{\texttt{FRAMEOBJECT}(f) : \texttt{pyObj}} \qquad \frac{f : list\ \texttt{pyObj} \rightarrow \texttt{valTyp}}{\texttt{FUN}(f) : \texttt{pyObj}} \qquad \frac{val : \texttt{valTyp}}{\texttt{createObj}(val) : \texttt{pyObj}}$$

$$\frac{\texttt{name} : str \quad \texttt{pid} : int \quad \texttt{value} : \texttt{valTyp} \quad \texttt{fields} : Map\ str\ \texttt{pyObj} \quad \texttt{methods} : Map\ str\ \texttt{pyObj}}{\{\texttt{name, pid, value, fields, methods}\} : \texttt{cls}}$$

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

$$\frac{l : list\ \texttt{cls}}{\texttt{LIST}(l) : \texttt{valTyp}} \qquad \frac{t : list\ \texttt{cls}}{\texttt{TUPLE}(t) : \texttt{valTyp}} \qquad \frac{d : list\ (\texttt{cls} * \texttt{cls})}{\texttt{DICT}(d) : \texttt{valTyp}} \qquad \frac{s : str}{\texttt{EXCEPTION}(s) : \texttt{valTyp}}$$

$$\frac{s : list\ \texttt{cls}}{\texttt{SET}(s) : \texttt{valTyp}} \qquad \ldots \qquad \frac{f : float}{\texttt{FLOAT}(f) : \texttt{valTyp}}$$

$$\frac{\varphi \triangleq \langle \Sigma_g, \Sigma_l, \Sigma_{l_+} \rangle : (Map\ str\ \texttt{pyObj} * Map\ str\ \texttt{pyObj} * list\ \texttt{pyObj}) \quad \Gamma : \texttt{codeObj} \quad i : int \quad \Delta : list\ \texttt{pyObj}}{\langle \varphi, \Gamma, i, \Delta \rangle : \texttt{frameObj}}$$

$$\frac{\Pi : list\ \texttt{bytecode} \quad \Sigma_c : list\ \texttt{pyObj} \quad \Sigma_v : list\ str \quad \Sigma_n : list\ str}{\langle \Pi, \Sigma_c, \Sigma_v, \Sigma_n \rangle : \texttt{codeObj}}$$

Fig. 2. Typing Rules for Python Objects

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

type valTyp = | INT: int -> valTyp
              | STRING: string -> valTyp
              | BOOL: bool -> valTyp
              | LIST: list cls  -> valTyp
                 ...
and cls = {
  name: string;
  pid: int;
  value: valTyp;
  fields: Map.t string pyObj;
  methods: Map.t string pyObj
}

and pyObj =
  | PYTYP: cls -> pyObj
  | CODEOBJECT: codeObj -> pyObj
  | FUN: (list cls -> builtins) -> pyObj
  | FRAMEOBJECT: frameObj -> pyObj
  | ERR: string -> pyObj
```

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

and frameObj = {
  dataStack: list pyObj;
  fCode: codeObj;
  pc: nat;
  f_localplus: list pyObj;
  f_globals: Map.t string pyObj;
  f_locals: Map.t string pyObj;
  f_cells: Map.t string pyObj;
  f_idCount: nat;
  f_usedIds: Map.t hashable nat
}

type vm = {
  callStack: list frameObj;
  code: codeObj;
  vmpid: nat;
  idCount: nat;
  usedIds: Map.t hashable nat
}
```

Fig. 5. Types in F★

# Evaluation

# Understanding Evaluation

- To understand how the Python's VM executes compiled bytecode, we used:
  - Python's bytecode documentations.
  - Investigated cpython source code whenever there were doubts.
- The challenge of english written documentations also appeared to exist in bytecode's documentations.
- cpython is is generally accepted as Python's reference implementation.

# Evaluation Rules

- The rules formalize how frames are evaluated and how the frame stack is managed.
- The frame stack has two state:

$$K \triangleright f \quad : \quad \text{Evaluation state}$$
$$K \triangleleft \mathsf{ret}(v) \quad : \quad \text{Return state}$$

- We start by *Evaluation State*:
  - During that state, the top frame $f$ is evaluated until it becomes $\mathrm{ret}(v)$.
- Once this happens:
  - Switch to *Return State,* which does one of the following:
    - Return the value v to the caller frame (top frame).
    - Spawn a new frame.
    - End evaluation and return v if the frame stack is empty **ε**.
- The evaluation of the frame stack uses the judgment $K \rightarrow K'$, where $K$ and $K'$ are frame stacks.
- The evaluation of frames uses the judgment $f \xmapsto{\Gamma.\Pi[i]} f'$, where $f$ and $f'$ are frames, and the arrow is labelled with the bytecode operation that is being executed.

# Frame Stack Semantics

- Describe how the frame stack is managed and how frames interact with each other (i.e. data flow between frames).

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

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

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

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

$$\frac{v \neq \mathsf{FRAMEOBJECT}(\_)}{\epsilon \triangleleft \mathsf{ret}(\langle \varphi, \Gamma, i, v :: \Delta \rangle) \longmapsto \mathsf{final}(v)} \tag{5}$$

Fig. 3. Instructions for managing the frame stack

# Frame Semantics

- Execute the code object inside a frame.
- Examples:

$$\frac{i < 0 \quad or \quad |\Gamma.\Pi| \leq i}{\langle \varphi, \Gamma, i, \Delta \rangle \xmapsto{\Gamma.\Pi[i]} \mathsf{ret}(\langle \varphi, \Gamma, i+1, \mathsf{ERR}(\text{“}\mathtt{Program\ counter\ is\ out\ of\ bounds}\text{”}) :: \Delta \rangle)}$$

$$\frac{}{\langle \varphi, \Gamma, i, \mathsf{ERR}(s) :: \Delta \rangle \xmapsto{\Gamma.\Pi[i]} \mathsf{ret}(\langle \varphi, \Gamma, i, \mathsf{ERR}(s) :: \Delta \rangle)}$$

$$\frac{}{\langle \varphi, \Gamma, i, v :: \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\mathsf{RETURN\_VALUE}} \mathsf{ret}(\langle \varphi, \Gamma, i, v :: \Delta \rangle)}$$

# Frame Semantics

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

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

$$\frac{\texttt{getMethod(floordiv,}v2\texttt{)} = \mathsf{FUN}(f) \quad f([v2, v1]) = \mathsf{VALTYP}(v') \quad u = \texttt{createObj}(\mathsf{VALTYP}(v'))}{\langle \varphi, \Gamma, i, v1 :: v2 :: \Delta \rangle \xrightarrow{\Gamma.\Pi[i]=\texttt{BINARY\_FLOOR\_DIVIDE}} \langle \varphi, \Gamma, i+1, u :: \Delta \rangle}$$

# Objects Extendability

$$\frac{\texttt{getMethod(floordiv},v2) = \mathsf{FUN}(f) \quad f([v2, v1]) = \mathsf{VALTYP}(v') \quad u = \texttt{createObj}(\mathsf{VALTYP}(v'))}{\langle \varphi, \Gamma, i, v1 :: v2 :: \Delta \rangle \xrightarrow{\Gamma.\Pi[i]=\mathsf{BINARY\_FLOOR\_DIVIDE}} \langle \varphi, \Gamma, i+1, u :: \Delta \rangle}$$

- What is $f$?

$$\frac{\texttt{obj} = \mathsf{PYTYP}(clsObj) \quad v = clsObj.value}{\texttt{getValue(obj)} = v}$$

$$\frac{\texttt{getValue(v1)} = \mathsf{INT}(\bar{v}1) \quad \texttt{getValue(v2)} = \mathsf{INT}(\bar{v}2) \quad \bar{v}1 \neq 0 \quad v' = \bar{v}2//\bar{v}1}{\texttt{floordiv([v2,v1])} = \mathsf{INT}(v')}$$

$$\frac{\texttt{getValue(v1)} = \mathsf{INT}(\bar{v}1) \quad \texttt{getValue(v2)} = \mathsf{INT}(\bar{v}2) \quad \bar{v}1 = 0}{\texttt{floordiv([v2,v1])} = \mathsf{ERR}(s)}$$

$$\frac{\texttt{getValue(v1)} = \mathsf{BOOL}(\bar{b}1) \quad \texttt{getValue(v2)} = \mathsf{INT}(\bar{v}2) \quad \bar{b}1 = false}{\texttt{floordiv([v2,v1])} = \mathsf{ERR}(s)}$$

$$\frac{\texttt{getValue(v1)} = \mathsf{BOOL}(\bar{b}1) \quad \texttt{getValue(v2)} = \mathsf{INT}(\bar{v}2) \quad \bar{b}1 = true \quad v' = v2}{\texttt{floordiv([v2,v1])} = \mathsf{INT}(v')}$$

Fig. 4. Formalization for the built-in type int

# 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 [Pierce 2002].

THEOREM 3.3 (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.*

LEMMA 3.2 (FRAME SEMANTIC PROGRESS). *A well-typed* `frameObj` *does not get stuck, that is, it is either in a return state or it can take a step according to the frame semantics rules.*

THEOREM 3.4 (PRESERVATION). *If a frame* $f$ : `frameObj` *evaluates to* $f'$, *then* $f'$ : `frameObj`.

# Formal Verification

- Py* implements these rules using one function for each bytecode instruction.
- These functions take as input the relevant frame components, and return the updated components.
- We deduce pre and post-conditions from the formal semantic rules:
  - Force them through the use of F*'s dependent typing system and Z3 (F*'s automated theorem prover).
- Example:

$$\frac{}{\langle \varphi, \Gamma, i, \beta, v :: \Delta \rangle \xrightarrow{\Gamma.\Pi[i]=\text{DUP\_TOP}} \langle \varphi, \Gamma, i+1, \beta, v :: v :: \Delta \rangle}$$

```
let dup_top datastack = (hd datastack)::datastack
```

```
val dup_top: (l:list pyObj {Cons? l}) -> Tot (l2:list pyObj {l2 == (hd l)::l})
```

# Discussion

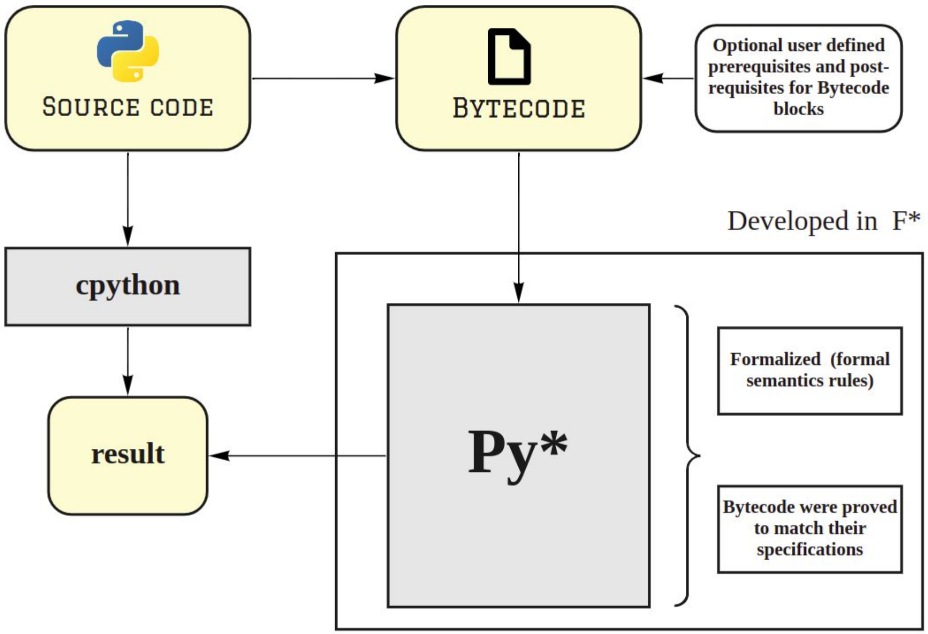
# Py* is a Reference

- The semantics of Py* could be used as a **reference** for other Python interpreter **implementations**.

- The techniques used for **formal verification** can be used by other **virtual machines** formalization projects.

# Finding bugs in other interpreters

- Py* could be used to find bugs in other Python interpreters.
- By running valid random Python code in Py* and other Python interpreters then observing results that don't match.
- We built an automated testing pipeline that we used for testing Py*, and we plan on extending it with comparisons with other VMs.



# Imporvements

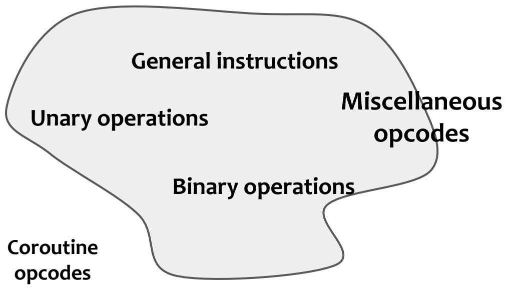
- We expect executing Speed to be lower than cpython.
  - Abstracting Imperative concepts in a functional environment is costly (E.g., hash tables).

- Could be solved by developing Py* in **Low*** instead of F*.
  - Low* have imperative concepts and code written in it can be translated into C code.
  - However, reasoning and formalizing such a thing will be much more difficult.

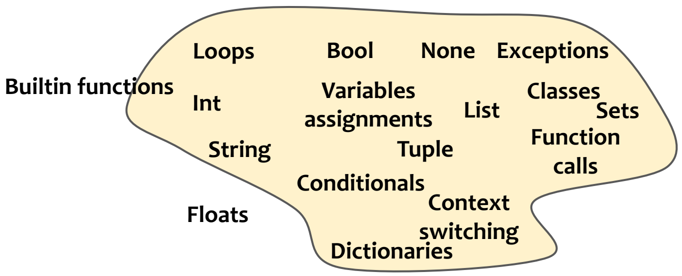
# Progress

- Defined formal semantics rules, implemented, formally verified, and extracted OCaml code for the following instructions within the shaped below.

## Bytecode instructions




## Python Source Code



# Future Work

- Support the rest of the bytecode instructions.
- Extend our test-suite to include all cpython's test-suite.
- Use the automatic testing pipeline to verify the correctness of different Python interpreters.

# Thanks!

***Any questions?***