Skeletal Semantics of a Fragment Python

JFLA 2024

Martin Andrieux Alan Schmitt

February 1st, 2024

Semantics in Skel

What is Skel

- Language for semantics description
- Focus on high-level constructs (unspecified functions)
- Non-deterministic programs
- Multiple backends (interpreter, debugger, Coq formalization)

Goals of this project

- Develop a readable and executable semantics of Python
- Stress-test Skel and speed up its development

Python's Challenges

No official specification or semantics

- Python Language Reference: just documentation
- CPython: Reference implementation

Unusual semantics

- Object-Oriented features
- Dynamic in many aspects

At our disposal

Python side

- Basic knowledge of Python
- Desugaring of Python in λ_π [Politz et al., 2013]
- Raphaël Monat's thesis, defining a semantics of Python

Skel side

- Team of Skel developers
- OCaml backend for interpreter derivation

Basics of Python Program Execution

- List of statements to execute
- Modifying a heap, an environment, and control flow

```
type stmt = ... (* Python AST *)
type list<a> = | Nil | Cons (a, list<a>)

val eval_stmt (s: stmt) = ...
val eval_program (p: list<stmt>) = ...

type addr
type heap (* addr -> values *)
type map<_>
type env = map<addr> (* string -> addr *)
type status = Cur | Ret | Exn | ...
```

A Simple Rule: Assign

$$\begin{aligned}&\mathbb{S}_{\text{cur}}\,[\![\texttt{id = expr}]\!]\,(\text{cur}, e, h) \overset{\text{def}}{=}\\ &\quad \text{letb}\,(\text{cur}, e, h), addr = \mathbb{E}\,[\![\texttt{expr}]\!]\,(\text{cur}, e, h)\ \text{in}\\ &\quad \text{return}\,(cur, e[\texttt{id} \mapsto addr], h)\end{aligned}$$

```
val eval_stmt (s: stmt) ((cur, e, h): (status, env, heap))
  : (status, env, heap) =
  match s with
  | SAssign (id, expr) ->
    let ((cur, e, h), addr) = eval_expr expr (cur, e, h) in
    let e' = write_env (e, id, addr) in
    (cur, e', h)
  end
```

Getting rid of the explicit state

```
type r = (local_env: env, builtins: builtins)

type s = ( global_scope: global_scope
         , heap: python_heap
         , scope_heap: scope_heap
         )

type flag = | Ret addr | Brk | Cont | Exn addr
type exn<a> = | Cur a | Flag flag

type m<a> = (r, s) -> (exn<a>, s)
```

Getting rid of the explicit state

```
type m<a> = (r, s) -> (exn<a>, s)

val return<a> (v : a) : m<a> =
  \(_, s):(r, s) -> (Cur<a> v, s)

val bind<a, b> ((w: m<a>), (f: a -> m<b>)) : m<b> =
  \(r, s):(r, s) ->
    let (vo, s') = w (r, s) in
    match vo with
    | Cur a -> let fa = f a in fa (r, s')
    | Flag f -> (Flag<b> f, s')
    end

binder @ = bind
```

Translation of a simple rule

$$\mathbb{S}_{\mathsf{cur}} \left[\!\left[\mathtt{id\ =\ expr} \right]\!\right] (\mathsf{cur}, e, h) \overset{\mathsf{def}}{=}$$

$$\mathsf{letb}\,(\mathsf{cur}, e, h), \mathit{addr} = \mathbb{E}\left[\!\left[\mathtt{expr}\right]\!\right] (\mathsf{cur}, e, h)\ \mathsf{in}$$

$$\mathsf{return}\,(\mathit{cur}, e[\mathtt{id} \mapsto \mathit{addr}], h)$$

```
val eval_stmt (s: stmt) : m<()> =
  match s with
  | SAssign (id, expr) ->
    let addr =@ eval_expr expr in
    write_env (id, addr)
  end
```

Scopes in Python

Why is it difficult?

- No explicit introduction of variables in a scope
- Nested functions and classes
- Scope indicators `nonlocal`, `global`

How to discover the semantics of scopes?

- Test with simple python programs
- Python reference
- Previous semantics ([Politz et al., 2013])

Guess the output!

```
x = 0
def function():

  x = 1
  print(x)

function(); print(x)
```

Guess the output!

```
x = 0
def function():

  x = 1
  print(x)

function(); print(x)
```

1 ; 0

Guess the output!

```
x = 0
def function():

  x = 1
  print(x)

function(); print(x)
```

1 ; 0

```
x = 0
def function():
  global x
  x = 1
  print(x)

function(); print(x)
```

Guess the output!

```
x = 0
def function():

  x = 1
  print(x)

function(); print(x)
```

```
1 ; 0
```

```
x = 0
def function():
  global x
  x = 1
  print(x)

function(); print(x)
```

```
1 ; 1
```

Guess the output!

```
x = 0
def function():

  x = 1
  print(x)

function(); print(x)
```

1 ; 0

```
x = 0
def function():
  global x
  x = 1
  print(x)

function(); print(x)
```

1 ; 1

```
def wrapper():
  x = 0
  def function():
    nonlocal x
    x = 1
    print(x)

  function(); print(x)

wrapper()
```

Guess the output!

```
x = 0
def function():

  x = 1
  print(x)

function(); print(x)
```

1 ; 0

```
x = 0
def function():
  global x
  x = 1
  print(x)

function(); print(x)
```

1 ; 1

```
def wrapper():
  x = 0
  def function():
    nonlocal x
    x = 1
    print(x)

  function(); print(x)

wrapper()
```

1 ; 1

Guess the output!

```
def function():
  print(x)


function()
```

Guess the output!

```
def function():
  print(x)


function()
```

NameError

Guess the output!

```python
def function():
  print(x)


function()
```

NameError

```python
x = 0
def function():
  print(x)


function()
```

Guess the output!

```
def function():
  print(x)


function()
```

NameError

```
x = 0
def function():
  print(x)


function()
```

0

Guess the output!

```
def function():
  print(x)


function()
```

NameError

```
x = 0
def function():
  print(x)


function()
```

0

```
x = 0
def function():
  print(x)
  x = 1

function()
```

Guess the output!

```
def function():
  print(x)


function()
```

NameError

```
x = 0
def function():
  print(x)


function()
```

0

```
x = 0
def function():
  print(x)
  x = 1

function()
```

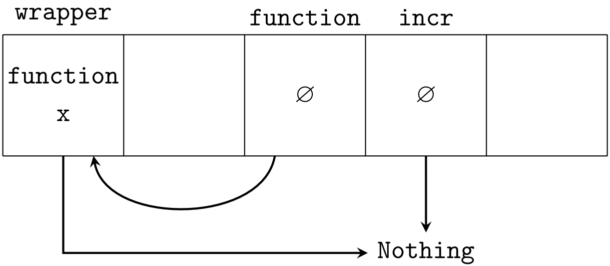
UnboundLocalError

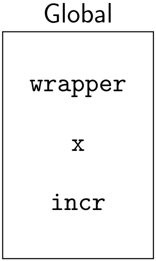
Things to remember

- Functions have a local scope, mapping identifiers to addresses
- Local variables are in the local scope, initialized to `LocalUndef`
- A variable is local if it is assigned (syntactic condition)
- A local scope is linked to its *upper* scope, for nonlocal variables
- The global scope is shared and total (no `LocalUndef`)

```
type env =
  | Global
  | InFun (map<var_scope>, scope_id)

type var_scope = | Local | NonLocal | Global
type scope_heap
```

Scope Heap





```
def wrapper():
  x = 0
  def function():
    nonlocal x
    x = 1
  return function
```

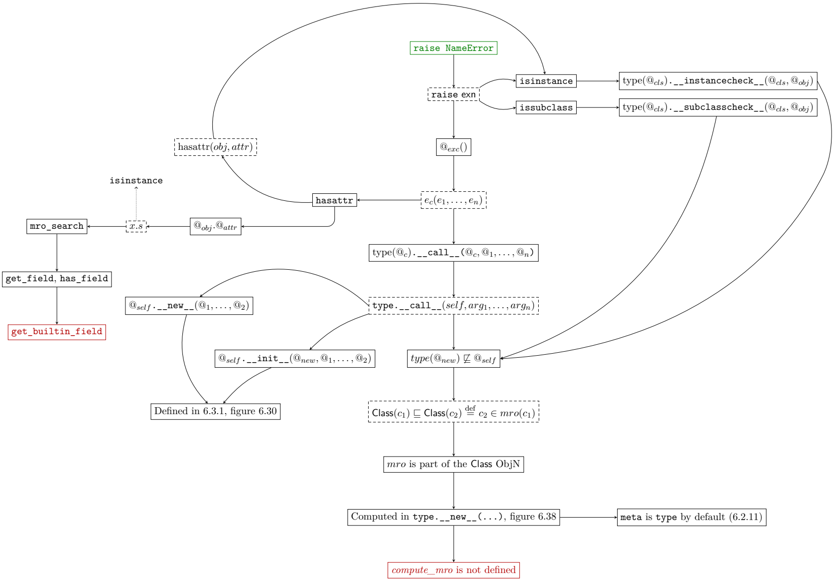
```
x = 0
def incr():
  global x
  x = x + 1
```

Scopes in Pyskel

```
type maybe<a> = | Nothing | Just a

type global_scope = map<addr>

type partial_addr = | LocalUndef | Value addr
type scope_info = (scope_id, map<var_scope>)
type partial_scope = (
  pscope_up: maybe<scope_info>,
  pmapping: map<partial_addr>
)

type scope_id = heap_addr<partial_scope>
type scope_heap = heap<partial_scope>
```

Semantics of Classes

- Well described in Monat's thesis, but without scopes
- Core feature of the language, need classes to implement classes
- Builtin functions are wrapped in classes
- Semantics is quite intricate



`raise NameError`

`isinstance`

type($@_{cls}$).`__instancecheck__`($@_{cls}, @_{obj}$)

`raise exn`

`issubclass`

type($@_{cls}$).`__subclasscheck__`($@_{cls}, @_{obj}$)

hasattr($obj, attr$)

$@_{exc}()$

`isinstance`

`hasattr`

$e_c(e_1, \ldots, e_n)$

`mro_search`

$x.s$

$@_{obj}.@_{attr}$

type($@_c$).`__call__`($@_c, @_1, \ldots, @_n$)

`get_field`, `has_field`

$@_{self}$.`__new__`($@_1, \ldots, @_2$)

`type.__call__`($self, arg_1, \ldots, arg_n$)

`get_builtin_field`

$@_{self}$.`__init__`($@_{new}, @_1, \ldots, @_2$)

$type(@_{new}) \not\sqsubseteq @_{self}$

Defined in 6.3.1, figure 6.30

$\mathsf{Class}(c_1) \sqsubseteq \mathsf{Class}(c_2) \overset{\text{def}}{=} c_2 \in mro(c_1)$

mro is part of the Class ObjN

Computed in `type.__new__(...)`, figure 6.38

`meta` is `type` by default (6.2.11)

compute_mro is not defined

Classes in python

Classes

- Set of definitions, called fields
- One or many superclasses
- Fields of a class + superclasses = attributes
- Few differences with function scopes, no `LocalUndef`
- Special methods such as `__init__`

Instances

- Reference to a class
- `__init__` method called at creation, if it exists

Guess the output!

```
class C:
  x = 0

  def f():
    return x

  def g(self):
    return self.x

C.x
```

Guess the output!

```
class C:
  x = 0

  def f():
    return x

  def g(self):
    return self.x

C.x
```

```
0
```

Guess the output!

```python
class C:
  x = 0

  def f():
    return x

  def g(self):
    return self.x

C.x
```

```
0
```

```python
C.f()
```

Guess the output!

```
class C:
  x = 0

  def f():
    return x

  def g(self):
    return self.x

C.x
```

```
0
```

```
C.f()
```

```
NameError
```

Guess the output!

```
class C:
  x = 0

  def f():
    return x

  def g(self):
    return self.x

C.x
```

0

```
C.f()
```

NameError

```
x = 'Hello'
C.f()
```

Guess the output!

```python
class C:
  x = 0

  def f():
    return x

  def g(self):
    return self.x

C.x
```

```
0
```

```python
C.f()
```

```
NameError
```

```python
x = 'Hello'
C.f()
```

```
’Hello’
```

Guess the output!

```
class C:
  x = 0

  def f():
    return x

  def g(self):
    return self.x

C.x
```

```
0
```

```
C.f()
```

```
NameError
```

```
x = 'Hello'
C.f()
```

```
’Hello’
```

```
C.g(C)
```

Guess the output!

```
class C:
  x = 0

  def f():
    return x

  def g(self):
    return self.x

C.x
```

```
0
```

```
C.f()
```

```
NameError
```

```
x = 'Hello'
C.f()
```

```
'Hello'
```

```
C.g(C)
```

```
0
```

Guess the output!

```
class C:
  x = 0

  def f():
    return x

  def g(self):
    return self.x

C.x
```

```
0
```

```
C.f()
```

```
NameError
```

```
x = 'Hello'
C.f()
```

```
’Hello’
```

```
C.g(C)
```

```
0
```

```
c = C()
c.x
```

Guess the output!

```
class C:
  x = 0

  def f():
    return x

  def g(self):
    return self.x

C.x
```

0

```
C.f()
```

NameError

```
x = 'Hello'
C.f()
```

'Hello'

```
C.g(C)
```

0

```
c = C()
c.x
```

0

Guess the output!

```
class C:
  x = 0

  def f():
    return x

  def g(self):
    return self.x

C.x
```

0

```
C.f()
```

NameError

```
x = 'Hello'
C.f()
```

’Hello’

```
C.g(C)
```

0

```
c = C()
c.x
```

0

```
c.x = 1
c.x; C.x
```

Guess the output!

```
class C:
  x = 0

  def f():
    return x

  def g(self):
    return self.x

C.x
```

```
0
```

```
C.f()
```

```
NameError
```

```
x = 'Hello'
C.f()
```

```
’Hello’
```

```
C.g(C)
```

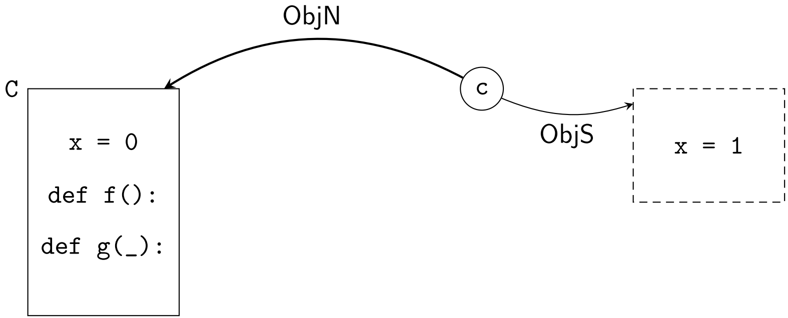
```
0
```

```
c = C()
c.x
```

```
0
```

```
c.x = 1
c.x; C.x
```

```
1; 0
```



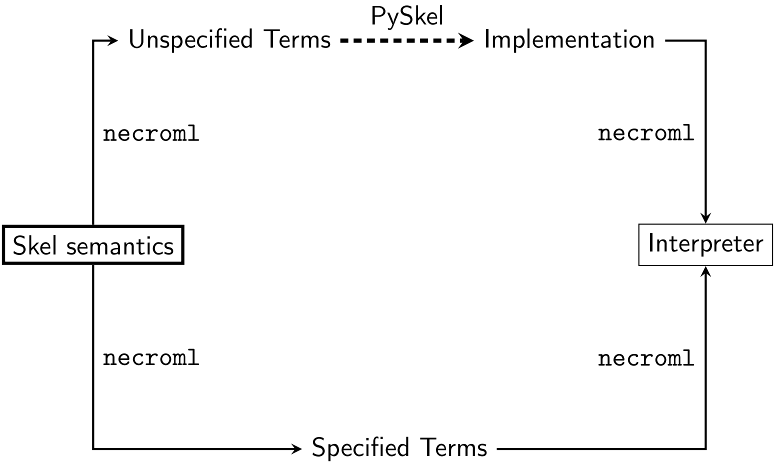
```
x = 0

def f():

def g(_):
```

```
x = 1
```

$$\mathcal{H} = \text{addr} \longmapsto \text{ObjN} \times \text{ObjS}$$

$$\text{ObjN} = \texttt{Int} \cup \texttt{Bool} \cup \texttt{Class} \cup \cdots \qquad \text{ObjS} = \{\texttt{Locked}\} \cup (\texttt{String} \longmapsto \text{addr})$$

Deriving an Interpreter



PySkel

Unspecified Terms

Implementation

`necroml`

`necroml`

Skel semantics

Interpreter

`necroml`

`necroml`

Specified Terms

Deriving an Interpreter

What is unspecified in Pyskel?

- Primitive types such as `int`, `bool` and `string`
- Values of these types, mostly strings like `"int"` or `"class"`
- Functions manipulating these types, like `internal_int_add`
- The heap and a polymorphic string map (for scopes)

- Simple stuff, around 100 lines of OCaml (against 1200 lines of Skel)
- $\approx$ 200 for parsing and functor instantiation

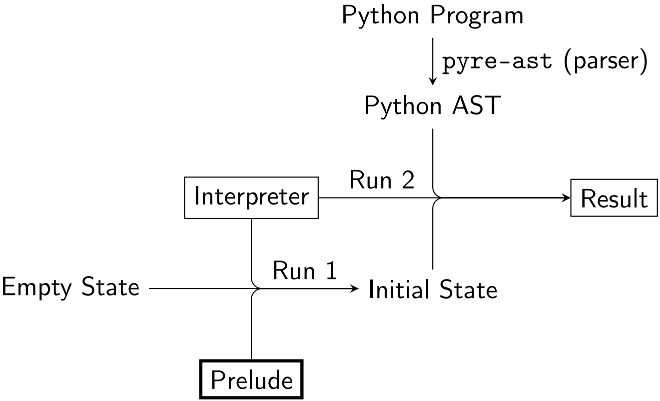
Builtin classes and methods

- The Python expression `a + b` is translated to `int.__add__(a, b)`
- Need for a class `int` in the initial state, with a method `__add__`
- Takes the form of a Python prelude file
- The add method must be a placeholder for Ocaml/Skel code

```
class int(object):
    @pyskel_internal("IntAdd")
    def __add__(self, other):
        pass
```

```
type internal = | IntAdd
val call_internal
  ((token: internal)
  ,(args: list<addr>)) : m<addr>
  = ...
```

Initial State



Python Program

`pyre-ast` (parser)

Python AST

Interpreter

Run 2

Result

Empty State

Run 1

Initial State

Prelude

Testing the semantics

undefined_in_function.py

```
# __result__

try:
    def f():
        x = y
        y = 0
    __result__ = f()
except UnboundLocalError:
    __result__ = True
```

Test

```
...

##########
# undefined in function
CPython : True
Pyskel  : True


...
```

Conclusion

Current status

- Base language with functions and classes (and scopes)
- Approach to add primitive operators
- Generation of an OCaml interpreter with simple tests

Future work

- Iterators and `for` loops
- Generators
- Necro debugger
- Coq generation