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 $\lambda_\pi$ [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

```plaintext
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

\[
S_{\text{cur}}[\text{id} = \text{expr}](\text{cur}, \text{e}, \text{h}) \overset{\text{def}}{=} \\
\text{letb}(\text{cur}, \text{e}, \text{h}), \text{addr} = E[\text{expr}](\text{cur}, \text{e}, \text{h}) \text{ in} \\
\text{return}(\text{cur}, \text{e}[\text{id} \mapsto \text{addr}], \text{h})
\]

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

```haskell
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

\[
\text{type } m\langle a \rangle = (r, s) \rightarrow (\text{exn}\langle a \rangle, s)
\]

\[
\text{val } \text{return}\langle a \rangle \ (v : a) : m\langle a \rangle = \\
\quad (\_, s):(r, s) \rightarrow (\text{Cur}\langle a \rangle \ v, s)
\]

\[
\text{val } \text{bind}\langle a, b \rangle \ ((w : m\langle a \rangle), \ (f: a \rightarrow m\langle b \rangle)) : m\langle b \rangle = \\
\quad (r, s):(r, s) \rightarrow \\
\quad \quad \text{let } (vo, s') = w \ (r, s) \text{ in} \\
\quad \quad \text{match } vo \ 	ext{with} \\
\quad \quad \quad | \ 	ext{Cur} \ a \rightarrow \text{let } fa = f \ a \ 	ext{in} \ fa \ (r, s') \\
\quad \quad \quad | \ 	ext{Flag} \ f \rightarrow (\text{Flag}\langle b \rangle \ f, s') \\
\quad \quad \text{end}
\]

\[
\text{binder } @ = \text{bind}
\]
Translation of a simple rule

\[ S_{\text{cur}} [id = \text{expr}] (cur, e, h) \overset{\text{def}}{=} \]

\[
\text{letb} (cur, e, h), \ \text{addr} = E[\text{expr}] (cur, e, h) \in\]

\[
\text{return} (cur, e[id \mapsto \text{addr}], h) \]

\[
\]

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!

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

function(); print(x)
```

```
1 ; 0
```

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

wrapper()
```

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

function(); print(x)

1 ; 0
Guess the output!

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

Output: 1 ; 0

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

Output: 1 ; 1

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

Output: 1 ; 1
Guess the output!

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

function(); print(x)
```

Output: 1 ; 0

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

function(); print(x)
```

Output: 1 ; 1

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

function(); print(x)
```

Output: 1 ; 1
Guess the output!

```python
def function():
    x = 1
    print(x)
function(); print(x)
```

1; 0

```python
def function():
global x
    x = 1
    print(x)
function(); print(x)
```

1; 1

```python
def wrapper():
    x = 0
    def function():
        nonlocal x
        x = 1
        print(x)
    function(); print(x)
wrapper()
```
Guess the output!

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

function(); print(x)
```

Output: `1 ; 0`

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

function(); print(x)
```

Output: `1 ; 1`

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

    function(); print(x)

wrapper()
```

Output: `1 ; 1`
Guess the output!

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

function()
```

Output:
```
NameError
```

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

function()
```

Output:
```
0
```

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

function()
```

Output:
```
UnboundLocalError
```
Guess the output!

def function():
    print(x)

function()  # NameError

x = 0
def function():
    print(x)
    x = 1
function()  # UnboundLocalError
def function():
    print(x)

def function():
    print(x)

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

def function():
    print(x)

x = 0

def function():
    print(x)

function()
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!

```python
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)

```haskell
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
```

Diagram:
- `wrapper`
  - `function` with `x`
  - `incr` with no variables
- `Global`:
  - `wrapper`
    - `x`
  - `incr`
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
- Built-in functions are wrapped in classes
- Semantics is quite intricate
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
class C:
    x = 0

def f():
    return x

def g(self):
    return self.x

C.x

c = C()
c.x; C.x

'Hello'

NameError

'Hello'
Guess the output!

class C:
    x = 0

def f():
    return x

def g(self):
    return self.x

C.x

C.x

0
class C:
    x = 0

def f():
    return x

def g(self):
    return self.x

C.x
C.f()
class C:
    x = 0

    def f():
        return x

    def g(self):
        return self.x

C.x

C.f()
NameError

C.g(C)
0

c = C()
c.x
0
c.x = 1
c.x; C.x
1; 0
class C:
    x = 0

    def f():
        return x

    def g(self):
        return self.x

C.x
0

C.f()
NameError

x = 'Hello'
C.f()
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 = C()
c.x
0
c.x = 1
c.x; C.x
1; 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
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
class C:
    x = 0

def f():
    return x

def g(self):
    return self.x

C.x

C.f()
NameError

c = C()
c.x

x = 'Hello'
C.f()

'Hello'

C.g(C)

0
class C:
    x = 0

def f():
    return x

def g(self):
    return self.x

C.x
0
C.f()
NameError
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

C.f()
'Hello'

C.g(C)
0

c = C()
c.x
0
c.x = 1
c.x; C.x
1; 0
Guess the output!

class C:
    x = 0

def f():
    return x

def g(self):
    return self.x

C.x
0

c = C()
c.x
0

c.x = 1
c.x; C.x
1; 0

C.f()
NameError

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

C.g(C)
0
Introduction

Python and Skel

Scopes

Object Oriented Features

Project Structure and Initial State

\[
\mathcal{H} = \text{addr} \rightarrow \text{ObjN} \times \text{ObjS}
\]

\[
\text{ObjN} = \text{Int} \cup \text{Bool} \cup \text{Class} \cup \cdots \\
\text{ObjS} = \{\text{Locked}\} \cup (\text{String} \rightarrow \text{addr})
\]
Deriving an Interpreter

- **Introduction**
- **Python and Skel**
- **Pyskel**
- **Scopes**
- **Object Oriented Features**
- **Project Structure and Initial State**

---

**PySkel**

- **Unspecified Terms** → **Implementation**
- **Skel semantics**
- **necroml**
- **Specified Terms**
- **Interpreter**
- **necroml**
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)
- ≈ 200 for parsing and functor instantiation
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:

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

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

Python Program
\[\text{pyre-ast (parser)}\]

Python AST

Interpreter

Run 1

Empty State

Run 2

Initial State

Result

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