Skeletal Semantics of a Fragment of Python

Martin Andrieux, Alan Schmitt

Skeletal Semantics of a Fragment of Python

Martin Andrieux[1] and Alan Schmitt[2]

[1]ENS Rennes, France
[2]INRIA, France

We present PySkel, a formalization of the semantics of a fragment of Python in Skel, a simple semantics description language. We describe a subset of the Python programming language including assignments, function calls, object oriented features, and exceptions. This subset is large enough to include challenges in the formalization of Python such as the handling of scopes. This formalization is used to generate an OCaml interpreter that can be used to run Python programs.

1 Introduction

The Python programming language is widely used, from teaching to research and industry. It is quite complex, however, for instance with its non-intuitive scoping rules. The closest artifacts to an official Python specification are the *Python Language Reference*, the *Python Standard Library*, and CPython, the reference implementation written in C. Unfortunately, neither of them are given as a formal semantics and only the latter is executable. For documentation and teaching purposes, and to provide the foundations for program analysis, it is most useful to have a formal and executable semantics of a language. We thus propose PySkel, a formal semantics of Python, written as a skeletal semantics [NS22] description, from which an OCaml interpreter can be extracted. Our work is based on Monat's denotational semantics of Python [Mon21].

The skeletal semantics description of a language can be thought of as an interpreter written in a strongly typed tiny functional language, called Skel. There are a few features that distinguish such a description from a usual interpreter. First, Skel allows for partial and non-deterministic programs with the `branch` construct. By partial, we mean that a Skel program can return no value, for instance with an empty set of branches. Next, it is not necessary to specify everything: some types or functions may be left unspecified, either because they are not crucial to the description of the language, because they are implementation dependent, or because they are specified later in an incremental definition. For instance, integers and associated operations such as addition are often left unspecified. Finally, Skel is a *syntax* to describe semantics. It is kept as simple as possible to be used with many tools, such as an OCaml interpreter generator, a debugger, or a Coq formalization generator.

Our contributions are a skeletal semantics of a fragment of Python that includes functions, classes, objects, and their complex scoping rules, as well as an OCaml interpreter derived from this semantics that can run Python programs and validate our semantics.

The paper is organized as follows. We introduce skeletal semantics in Section 2. We then focus on two challenging features of Python and their description in Skel: scopes in Section 3 and object-oriented aspects in Section 4. We show how to derive an interpreter in

```
type addr, heap, id
type expr
type stmt = | SAssign (id, expr)

val eval_expr : (heap, expr) -> (heap, addr)
val write : (heap, id, addr) -> heap

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

Figure 1. A Simple Skeletal Semantics

Section 5 and in particular describe how we validate our approach in Section 5.3. We discuss related work in Section 6 and conclude in Section 7. Our formal semantics and interpreter are available online.[1]

2 Monadic Skeletal Semantics

We first describe Skel, the language of skeletal semantics, and how it leverages monadic features to model complex effectful computations in its simple functional language while keeping readability. We shortly describe the features we need for PySkel, a longer introduction to Skel and monads is available in [KS].

2.1 Skeletal Semantics

Consider the Python statement `x = e`. To provide the semantics of this statement, we write a Skel program containing an evaluation function `eval_stmt`, which takes a statement and other arguments such as the heap as input. A toy version of this Skel description is presented in Figure 1.

In this example, we start by declaring types. Some of them are unspecified (they do not have a definition): `addr`, `heap`, `id`, and `expr`. The first three are considered implementation-dependent, while the `expr` type for expressions is specified later in this paper. The `stmt` type is specified as a data type with a single constructor in this example. Note that it may depend on unspecified types, as is the case here. Then, we declare two unspecified functions `eval_expr` and `write` by giving their types. These functions cannot be specified as they manipulate data of unspecified types. Finally, we provide a specified function, `eval_stmt`, which inspects the statement using pattern matching and modifies some state.

To go in more details, the `eval_stmt` function takes as arguments a heap and a statement. It pattern-matches the statement, and in the (only) case it is an assignment of `id` to `expr`, it executes the relevant code. First, it evaluates `expr`, returning a new heap `h'` and an address. Indeed, as in Monat's work [Mon21], the evaluation of an expression returns a heap address. We then modify the heap according to the assignment and return it. All these steps are sequenced using the `let p = e1 in e2` operator, which evaluates `e1`, matches its result against `p`, then evaluates `e2`.

This simplified code (which does not include local environments nor control-flow issues such as returning from a function call or raising an exception) is already quite cumbersome, as we explicitly manipulate the heap at every step. Adding exceptions requires that we additionally check at every evaluation step that the returned address is not an exception,

[1] https://gitlab.inria.fr/skeletons/pyskel/-/tree/jfla2024

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

type s = ( heap: python_heap
         , global_scope: global_scope
         , scope_heap: scope_heap (* functions (shared) *)
         , scope_stack: list<map<addr>> (* classes (not shared) *)
         )
type r = ( local_env: env
         , builtins: builtins
         )

type m<a> = (r, s) → (exn<a>, s)

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

val bind<a, b> ((w: m<a>), (f: a -> m<b>)) : m<b> =
  λ(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
```

Figure 2. Reader, State, and Exception Monad

and adding environments requires a new argument to the evaluation function. To deal with these issues, we show in the next section how to transform this code in a monadic version that seamlessly propagates such effects.

2.2 Monadic Style

The use of monads to add effects to a functional programming language is well-known [Wad90]. Skel, having first-order functions and polymorphic types, can be easily equipped with monads. We use some syntactic sugar to make the resulting code easier to read.

The monad we use in our semantics is given Figure 2. It combines three monads: a *reader monad* to give read-only access to a local environment and to built-in values (record type `r`), a *state monad* for the read-write access to the heap and the scopes (record type `s`), and an *exception monad* indicating whether the result is a normal value or not (algebraic type `exn<a>`, detailed below).

More precisely, the monadic type we consider is `m<a>`, where `<a>` is a polymorphic annotation. This type for computations is defined as `(r, s) → (exn<a>, s)`: a computation takes a current environment and state as input, and it returns a possibly exceptional result and a new state. The type `exn<a>` is either `Cur a`, an actual result of type `a`, or `Flag flag`, indicating a control-flow breaking result. This flag can signal the return of a function `Ret addr`, where `addr` is the returned value, the breaking out or continuing of a loop, and an exception `Exn addr` where `addr` is the exception object.

```
val eval_expr : expr → m<addr>
val write : (id, addr) → m<()>

val eval_stmt (s: stmt) : m<()> =
  match s with
  | SAssign (x, v) →
    let addr =@ eval_expr v in
    write (x, addr)
```

Figure 3. Monadic Skeletal Semantics (excluding types)

The monad associated to this computation type has the usual two functions. The function `return<a>` takes a pure value of type `a` and puts it in the monad as a computation of type `m<a>`. Its code, depicted in Figure 2, is the anonymous function `λ(_, s):(r, s) → (Cur<a> v, s)` that takes a pair `(_,s)`, ignoring environment and naming the state `s`. It then returns a normal result `Cur<a> v` alongside the unchanged state `s`. This code shows that constructors of polymorphic types must be annotated with the instantiation type. The function `bind<a,b>` takes a computation `w` of type `m<a>`, a continuation `f` of type `a → m<b>` (given a pure value of type `a`, return a computation of type `m<b>`), and it chains them together to return a computation of type `m<b>`. To this end, it first takes the current environment and state (`λ(r,s):(r,s) → ...`). It then runs the computation `w` by giving it the environment and the state, which returns a potentially exceptional value `vo` and a new state `s'`. If the value is a pure value `Cur a`, then the continuation is called with it, passing along the initial environment and the modified state.[2] If the result is a flag, then the continuation is not called and the flag is directly returned with the modified state. Note that the type of the flag is changed to the expected output type, as required by our strongly typed language.

To streamline notations, a symbol can be associated to a binder (`binder @ = bind`), so that code of the form `bind e1 (λv → e2)` can be simply written `let v =@ e1 in e2`. This is similar to OCaml's `(let*)` and Haskell's leftarrows in `do`-notation. The resulting Skel code is given in Figure 3, where the result of the `eval_stmt` function lives in the `m<()>` monadic type, i.e., computations that return unit.

2.3 Skeletal Description of Python Semantics

The skeletal description of the fragment of Python we consider is mostly contained in the `skeletal.sk` file. This file contains six main sections: the syntax of Python, the declaration of builtins, the definition of the state, environment, and flags types, the monad used by the interpreter, the read and write functions that deal with scopes, and finally the evaluation functions themselves.

The semantics of Python is quite usual, but some points deserve special attention. In particular, scope management differs from most functional languages (Section 3). We also focus on the object oriented features (Section 4) and the management of the initial state of the interpreter (Section 5).

3 Scopes

We describe in this section the way scopes are managed in Python and its implementation in Skel. We focus on functions in this section, the scopes for classes have some further subtleties that are detailed in Section 4.

[2] Skel is so simple that it does not have n-ary application, hence the application of `f` to `a` and `(r,s')` must be done in two steps with a let-binding.

```
x = 0

def foo():
  y = 1
  def bar():
    z = 2
    print(z)
    print(y)
    print(x)
  bar()

foo()
```

```
x = 0

def foo():
  x = 1
  print("foo:", x)

foo()
print("glb:", x)
```

```
x = 0

def foo():
  global x
  x = 1
  print("foo:", x)

foo()
print("glb:", x)
```

```
def foo():
  x = 0
  def bar():
    nonlocal x
    x = 1
    print("bar:", x)
  print("foo:", x)
  bar()
  print("foo:", x)

foo()
```

Figure 4. Scopes in Python

3.1 Scopes in Python

Scopes in Python are lexical: they correspond to the embedding of function declarations. The outermost scope is the *global scope*. When accessing a variable, the local scope is first explored, then the surrounding scope, up until the global scope. This is illustrated by the first program of Figure 4 which prints in order `2`, then `1`, then `0`.

In Python, there is no notion of variable declaration to state that a variable should be added to a scope. The first syntactic occurrence of an assignment to a variable corresponds to a declaration of this variable in the current scope. To illustrate this, consider the second program of Figure 4. The inner function has a local `x` which is set to `1`, so the scope of `foo` has this variable. Hence, the program first prints `foo:1` for the local `x` when `foo` is called, then `glb:0` for the global (different) `x`.

To tell Python that an assignment is not a variable declaration, one should state the scope of the variable before any assignment, so that this assignment does not declare the variable. One may state that a variable is global, as shown in the third program. This program prints `foo:1` and `glb:1`, illustrating that the assignment in `foo` did change the global `x`, hence they correspond to the same variable. Note that the global annotation for a variable also impacts how it is read. The following code for instance outputs `3`. Indeed, as `x` is not defined in `baz`, it is looked up in `bar`, where it is said to be global, thus bypassing the declaration in `foo`.

```
x = 3
def foo():
    x = 1
    def bar():
        global x
        def baz():
            print(x)
        baz()
    bar()
foo()
```

Alternatively, a variable assignment may refer to an enclosing scope that is not the global scope. This is illustrated in the fourth program of Figure 4, where `x` is local to the function `foo`. In the enclosed function `bar`, `x` is declared as `nonlocal`, which means it must be declared in an enclosing function. It does not have to be the immediately enclosing function, but it cannot be the global scope. When executed, this program prints `foo:0`, `bar:1`, and `foo:1`.

Note that the `global` and `local` keywords only matter for determining which variable to access and that they are syntactic constructs. For instance, the variable written by an assignment is statically known when parsing the program. However, with the exception of the global scope, scopes are created at run-time (typically when calling a function), and they may persist after the function's completion. Consider code of Figure 5. After calling

```
def counter(init):
  x = init
  def reset():
    nonlocal x
    x = 0
    print(x)
  def incr():
    nonlocal x
    x = x + 1
    print(x)
  return(reset,incr)

r,i = counter(5)
i(); r(); i()
```

Figure 5. Lingering Scopes

`counter`, the two returned functions share its scope as enclosing scope, where the nonlocal variable `x` is defined: the scope still needs to be present after the call to `counter` is finished as `r` and `i` may be called. The code thus prints `6`, `0`, then `1`.

To summarize, the *global scope* contains all the toplevel declarations, it is a mapping from variable names to heap addresses. Calling a function creates a new scope containing the parameters and the local variables of the function. Functions may need to access enclosing scopes to read or write non-local variables. We depict this dependency as a tree of scope, as shown in Figure 6. The dashed arrows to the global scope are only followed when reading variables that are not defined locally. In fact, the scopes of the `incr` and `wrapper` functions do not have an upper scope. We show on the right of each scope the names that are defined in this scope.

```
def wrapper():
  data = # some data
  def worker():
    nonlocal data
    # work
  return worker()

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

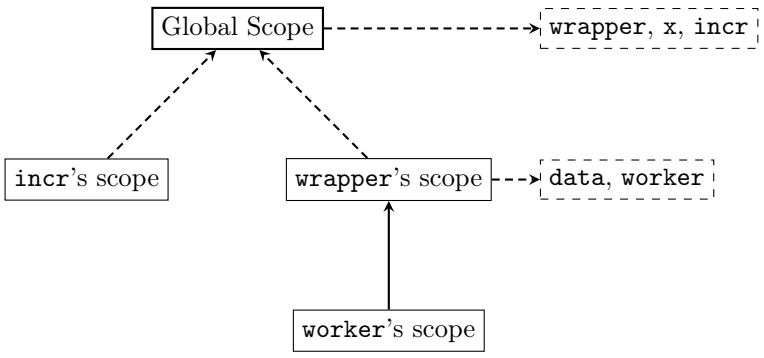

Figure 6. Simple Functions and Corresponding Scopes

Note that there is one scope per function *call*. The diagram in Figure 6 is valid if `wrapper` and `incr` have been called exactly once. The local variables are computed at parsing time, and are allocated at run time when the function call occurs. It is thus possible to read an allocated but not initialized variable. Such a read will raise `UnboundLocalError`, instead of the `NameError` exception that is raised when a variable does not exist. This is illustrated by the following code:

```
def f():
  x = undef  # raises NameError as undef does not exist

def g():
  x = y  # raises UnboundLocalError because y is local to g but not yet bound
  y = 0
```

3.2 Scopes in PySkel

PySkel extends Monat's semantics with the previously described scopes. We distinguish two types of scopes, each representing a mapping from identifiers to values, but with some special properties.

First, we define a global scope, containing the top level functions and values. The global scope is unique and shared between all code, with a simple structure. In Skel, it is defined as the declaration `type global_scope = map<addr>`, i.e., a mapping from strings to adresses (see below). It is part of the mutable state described in Figure 2.

We next introduce the type of function scopes. As these scopes may contain uninitialized values, we define a `partial_addr` type, isomorphic to Haskell's well-known `Maybe` type,[3] with a `LocalUndef` constructor. In addition, function scopes can be shared and can access upper scopes. To account for this, we define an additional heap for scopes, so scopes are allocated and referenced with *scope identifiers*.

Scopes are a data structure to organize identifiers, but they are not sufficient. For example, to execute a variable assignment, one should know in which scope to look for it (i.e., is the variable local, global, or nonlocal). To this end, scopes are accompanied by a map from variable names to the kind of scope they should be searched in.

```
type var_scope = Local | Nonlocal | Global
```

This map is built as follows: global and nonlocal variables are explicitly given, and the remaining assignments are local variable declarations. So we need two pieces of information to manipulate variables in a function: the scope and the location map.

A function scope is thus a pair of a map from strings to `partial_addr` for the scope contents and an optional `scope_info` for the enclosing scope. The final type is presented below.

```
type heap<_> (* Unspecified heap, heap<a> contains values of type a *)
type heap_addr<_> (* heap_addr<a> are the addresses of heap<a> *)

type map<_> (* Unspecified map, map<a> is a mapping from strings to a *)
type maybe<a> = | Nothing | Just a

type partial_addr = | LocalUndef | Value addr
type var_scope = | Local | NonLocal | Global

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

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

The `var_scope` map is not part of the `partial_scope` type. This is because the map is read-only in a function, and we thus avoid to store it in the mutable state.

Note that the `heap` type is parameterized while the one presented in Figure 1 is not. As a consequence, we need to specify the type of values stored in the main Python heap. It is well described in Monat's semantics (the two sets ObjN and ObjS, presented in Chapter 6 of his thesis), and can be directly translated to Skel. Heap values are composed of two elements, respectively of type `obj_n` and `obj_s`. The `obj_n` type represents the different kinds of values (`Int`, `String`, `Fun (..)`, etc). The `obj_s` part is a dictionary containing the fields of a value, with a special value `Locked` for primitive values. We thus define the `heap` type as follows.

```
type obj_n = | Int python_int | String python_string | ...
type obj_s = | Locked | Fields map<addr>
```

[3] We could also use the `maybe` type from PySkel, but we define a new one to clarify the code.

```
type addr = heap_addr<(obj_n, obj_s)>
type python_heap = heap<(obj_n, obj_s)>
```

Note that the `obj_n` type contains a constructor `Fun` for functions, this constructor includes a `scope` field indicating the scope where the function has been defined, i.e., its enclosing scope when called.

3.3 Function Calls and Memory Access

We now summarize the different actions performed to evaluate a function. The goal of this part is to provide a global understanding of functions and to justify the design of scopes.

3.3.1 Function Declaration

Evaluating a function declaration statement consists in the allocation of a `Fun` object on the heap. It is comprised of the name of the function, the list of its arguments, the list of its local variables, the code of its body, and the optional enclosing scope, which is set to the current scope unless the function is declared at top level.

3.3.2 Function Call

To call a function, we first extract the `Fun` object from the heap. We create a fresh scope containing the arguments of the function (bound to the values given in the function call) as well as the local variables (bound to `LocalUndef`). We set the `pscope_up` field of the fresh scope to the scope information stored in the `Fun` object.

Then, the variable map `var_map` is computed by inspecting the body of the function. It is currently not stored in the `Fun` object to keep the `obj_n` type close to Monat's semantics. Once this is done, we evaluate the body with the new scope information, by locally setting the environment to `InFun (fun_scope, var_map)`.

The environment, first declared in Figure 2, keeps track of the evaluation context. This context can have three forms: `Globl`, `InFun`, or `InClass`. `Globl` means "in the global scope" or "at top level". It does not provide any additional information. The `InFun` state contains a `scope_info` and represents the evaluation of a function. The `InClass` is described in the next section. We need to be able to distinguish between contexts because the scoping rules are different in a function, a class, or at toplevel.

3.3.3 Memory Access

To read or write a variable in a function, we first look at the `var_scope` associated to the variable to know if it is local, nonlocal, or global. We can then look for the variable in the right scope, knowing that the identifier of the local scope is in the read-only state, the upper scope is stored in the local scope, and the global scope is stored in the mutable state.

4 Objects and Classes

Classes constitute a major aspect of Python. They are well described in Monat's semantics, with the exception of the subtle interactions with function scopes. In this section, we first introduce the basic aspects of classes and objects in Python before detailing the scope issues and our implementation in Skel.

4.1 Classes in Python

A class defines a new *type*, consisting of a set of *fields*, declared in the class body. These fields include variables and function definitions. Classes can inherit from each other, the set of fields of a class and its super classes forms the *attributes* of the class.

The set of fields is given as a list of statements, just like a function body. It may contain global and non-local variables, declared with the `global` and `nonlocal` keywords. The variables so defined are not part of the fields of the class.

The notion of field is hidden in Python, but it is nevertheless necessary to understand the underlying mechanism of classes. In the code sample given in Figure 7, the class `Example` has three fields: `value`, `function`, and `method`. As the class inherits from `Super`, the set of attributes may be larger than the set of fields.

```
class Example(Super):
  value = 0
  def function():
    return 1

  def method(self, value):
    self.x = value
```

Figure 7. A Class in Python

To create an instance of some class, we call the class like a function: `Example()`. This creates a reference to the class called. Instances share the attributes of the superclass. These attributes can be accessed with the dot operator, as well as attributes of the instance itself: `Example().value`. Such access can have some side effects. In particular, if an instance accesses a function of its class, a new *method* is allocated and returned instead of the function. This method contains two addresses: the instance (i.e., calling object) and the function. Thus, the method call simply passes the instance as first parameter to the function. The given instance is modified according to the function body defined in the class. This mechanism is illustrated in Figure 8.

```
class Example:
    def method(self, value):
        self.x = value

instance = Example()

Example.method # is a function
instance.method # is a method, referencing `instance` and `Example.method`
instance.method(2) # is nearly the same as
Example.method(instance, 2) # "nearly" because no method is allocated here
```

Figure 8. Functions and Methods

As can be seen, instances can have additional fields that are not part of the class (`x` in the example). Those fields are stored on the heap, next to the object they belong to (as the structural part ObjS of the value representing the object). Note that some builtin object, like integers, are *locked*: they do not contain any such extra field. Trying to access or set them, as in `x = 1; x.y = 2` results in an attribute error.

As classes are well described in Monat's semantics, they are not studied any further in this article. However, we have also implemented nonlocal and global scopes in functions, so we need to explain how classes are affected by this addition.

4.2 Classes and Scopes in PySkel

In PySkel, the representation of classes contains a map from identifiers to addresses to store the values of the fields. This is the only non-trivial part of the evaluation of a class definition. The bodies of classes are similar to the ones of functions, including the declaration of `global` and `nonlocal` variables. A first approach for their evaluation would consist of allocating a

fresh *function* scope and evaluating the body in it, to populate the scope, hence to obtain the values of the fields of the class. We do something very similar in PySkel, the only difference is the scope type: functions scopes are designed for functions and are not adapted to class evaluation. To understand why, we next clarify the differences between classes and functions.

Firstly, class variables cannot be uninitialized, hence we do not need partial maps. Secondly, fields exist only in the class, they are not directly accessible from functions or classes defined inside the class (methods may still access them as fields of their `self` argument). Thus the scope chain need to bypass classes (see the body of the `bar` function in Figure 9, where the closest non-class scope where `x` is defined is the scope of `foo`). Finally, class scopes cannot be shared (as mutable fields only exist in the instance), so there is no need to store them on the heap.

```
def foo():
    x = 0
    class C:
        x = 1
        def bar():
            print(x)
    return C

foo().bar() # 0
```

Figure 9. Scopes Interaction Between Functions and Classes

Hence, we need a simple address map, like the global scope, with some upper scope for nonlocal variables. In case of nested classes, we may need to pause the construction of the current class scope to evaluate the subclass. This is why we have a `scope_stack` in the mutable state. At any point of the program, the length of the stack corresponds to the number of classes we entered to reach the program point. Once the body of the class is fully evaluated, we pop the scope from the stack to obtain the expected dictionary.

The final type of the environment is defined below. The `Globl` and `InFun` constructors have been introduced in the previous section. For classes, the context is composed of the `var_scope` map and the upper scope information. The current scope is at the top of the stack, hence we do not need a scope identifier.

```
type env =
  | Globl
    (* var_scope and current scope *)
  | InFun scope_info
    (* var_scope and scope_up *)
  | InClass (map<var_scope>, maybe<scope_info>)
```

5 Deriving an Interpreter

Once the semantics is written, we can use the various Skel backends to test and demonstrate the usability of the semantics. We describe in this section the process of deriving a Python interpreter using the OCaml generator backend, leaving the use of the Coq and debugger backends for future work.

5.1 Project Structure

The Skel toolchain contains a program called `necroml`. It takes a Skel semantics and translates it to an OCaml interpreter. The generated code takes the form of a functor

requiring an implementation of the unspecified types and terms. PySkel uses `necroml` to derive a simple Python interpreter which is used to test the semantics. For a better understanding of the PySkel project structure, we provide a summary diagram in Figure 10.

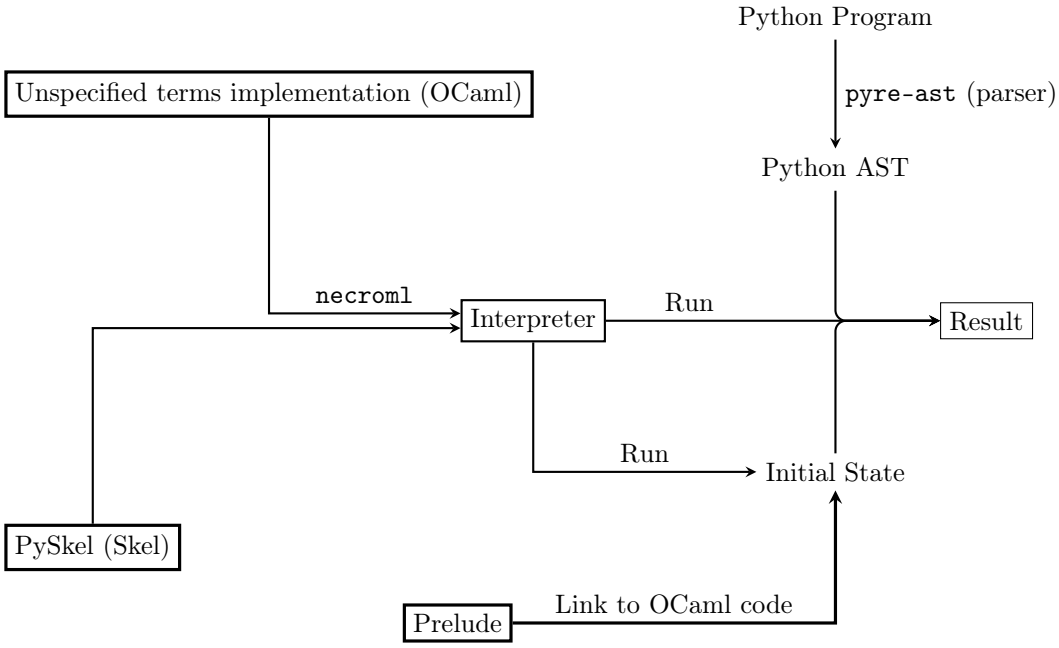

Figure 10. Structure of the PySkel Project

The `necroml` program generates specified terms and requires an implementation of unspecified ones. This implementation is written in OCaml and it is part of the PySkel project. With these two parts, a functor instantiation gives the Python interpreter. We can run this interpreter by providing a program and some initial state (which corresponds to the state part of the monad). This state is computed from a prelude file containing the builtin classes (such as exceptions) and some glue to fit well with the Skel semantics (see Section 5.2). Python programs are parsed with the `pyre-ast` OCaml library and then translated to the PySkel AST.

The unspecified part of PySkel is small. It is composed of primitive types (`bool`, `int`), functions that operate on these types, the heaps, and a polymorphic string map. The reason we chose to leave these parts unspecified is pragmatic: they are well-known basic features whose behavior does not need to be clarified in Skel. Note that if this came to change, one could simply replace the unspecified declarations with their chosen specification. The implementation of the unspecified parts is 100 lines long, compared to the 1200 lines of Skel (the instantiation files can be found in the `lib` directory, they are then used at the beginning of `utils.ml` to generate the interpreter).

5.2 Initial State and Internal Function

The evaluation of a Python program requires some initial state. This state must contain builtin objects, functions, and classes.

As the organization of builtin functions are not really part of Python evaluation mechanism, we do not want to specify this organization in the semantics. Note that functions are defined in the semantics, but not organized in classes. For example, the code to add integers is written in the Skel file, but not linked to any `int` class or `__add__` method. Hence, we need a way to organize existing Skel code into classes from outside the semantics.

This is done with a Python `prelude.py` file, containing declaration of classes and methods. We provide a way to relate it to Skel code with a decorator `@pyskel_internal`, as shown in Figure 11. The writer of the prelude file must specify a token name (`"IntAdd"` in the example) to identify the corresponding code. We can understand this as "when you try to call this function, call the Skel code associated to the `IntAdd` token instead". Then, in the semantics, the `call_internal` function redirects the call to execute the desired code, as shown in Figure 12. The purpose of this code is to specify the number of arguments, the order in which they are evaluated, the type of the arguments, and the behavior to follow if the builtin requirements are not met. As these are semantic aspects, we prefer to specify them directly in the code in Skel.

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

Figure 11. Declaring an Internal Function in a Python Class

```
type internal = | IntAdd

val call_internal ((internal : internal), (args : list<addr>)) : m<addr> =
  match internal with
  | IntAdd ->
     match args with
     | Cons (op1, Cons (op2, Nil)) ->
       let (objn1, _) =@ read op1 in
       let (objn2, _) =@ read op2 in
       match (objn1, objn2) with
       | (Int left, Int right) ->
          let result = internal_int_add (left, right) in
          alloc (Int result, Fields map_empty<addr>)
       | _ ->
          let r =@ ask in
          ret r.builtins.notImplemented
       end
     | _ ->
        raise_exn<addr> _TypeError
     end
  end
```

Figure 12. `call_internal` Function in the Skel Semantics

The connection between the decorator and the internal call is done during the translation from the Pyre AST to the Skel AST.

5.3 Using the Derived Interpreter to Test the Semantics

Once the interpreter is derived, we are able to execute Python programs. The test suite of PySkel consists of a set of Python files, executed twice: once with the official CPython interpreter and once with PySkel. We then check if the results are the same. Any inconsistency means we need to correct our semantics, as the C interpreter of Python is considered as ground truth. We test semantics aspects as well as the construction of the initial state in the files found in the `class`, `exception`, `internal`, and `scope` directories. In addition to having a semantics that is very close to the one described by Monat, this gives us confidence to believe that our interpreter correctly evaluates Python programs, hence it can be used as a description of Python's semantics.

6 Related Work

There have been several proposals to formalize the semantics of Python. Monat gives a denotational semantics (on paper) in his PhD thesis [Mon21], which is used to build analyzers as part of the MOPSA platform [MOM21]. His semantics is quite faithful as it is derived from the official Python interpreter written in C. Our work is heavily based on this semantics, extended to deal with additional features absent from Monat's thesis, such as nonlocal and global scopes, both for functions and classes. We also provide an interpreter, whereas Monat's semantics is not executable. In addition, our Python semantics is written in Skel, so it can easily be reused for other tools.

An initial attempt to formalize the semantics of Python was done by Politz et al. as a small-step operational semantics [PMM+13]. This semantics is actually given as a translation, called *desugaring*, of Python in λ_π, a much smaller core language. The desugaring process and an interpreter for λ_π are both written in Racket [FFF+18]. When combined, they provide an interpreter for Python source code. We have relied on this work to grasp the subtlest aspects of Python's scopes, in particular for classes defined inside functions. The distinguishing feature of our work compared to theirs is that we describe the semantics directly at the source code level. The desugared semantics can be quite different from the source, especially when it is CPS-transformed to deal with generators. Understanding the semantics of a Python program by looking at an execution of its desugared version may be challenging. Although we have not yet implemented generators, we believe we can seamlessly handle them, as discussed in Section 7.

Finally, some recent work aims at giving a semantics of Python's bytecode through a formalization in F* [Kar22]. Once again, it is challenging to reason about a source program through the semantics of a translation, here into bytecode.

7 Conclusion and Future Work

We have presented PySkel, a formalization of the semantics of a fragment of Python in Skel, which we have used to derive a Python interpreter in OCaml. The development of this complex semantics has helped identify an area of improvement for Necro, mainly the inclusion of files, which is now available in a recent version of the tool. Although our formalization covers only a tiny subset of the standard library, we believe it is illustrative enough to be easily extended. We plan to add `for` loops next, as they pose some interesting questions regarding scopes and are not directly formalized in Monat's thesis, before considering generators.

Generators can be seen as a way to interrupt the execution of a program, yielding an intermediate value and a computation to resume. They are quite complex to capture formally as they require handling partial computations and their continuations. Our plan to model their semantics in Skel is to reuse the work of Khayam et al. [KS] to use a delimited continuation monad. More precisely, they have shown that, since Skel is almost in administrative normal form [SF92], one may choose an appropriate monad to seamlessly capture side-effects, including the interruption of a computation because of a `yield`. The integration of their approach, only tested on a toy language, in this much larger semantics would challenge how seamless the approach is.

Finally, it would be most useful to be able to run CPython's test suite, to more thoroughly test our work. Unfortunately, this test suite depends on the `unittest` framework, which uses native libraries and reflective features on modules. Adding these to PySkel is a long term goal.

References

[FFF+18] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, 2018.

[Kar22] Ammar Karkour. Py*: Formalization of Python's Verifiable Bytecode and Virtual Machine in F*, 7 2022.

[KS] Adam Khayam and Alan Schmitt. A practical approach for describing language semantics. https://people.rennes.inria.fr/Alan.Schmitt/papers/programming_draft.pdf.

[MOM21] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In *Static Analysis Symposium (SAS)*, pages 1–23, Chicago, Illinois, United States, October 2021.

[Mon21] Raphaël Monat. *Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries*. PhD thesis, Sorbonne Université, 2021.

[Noi] Louis Noizet. Necro Debugger Generator, https://gitlab.inria.fr/skeletons/necro-debug.

[NS22] Louis Noizet and Alan Schmitt. Semantics in Skel and Necro. In *ICTCS 2022 - Italian Conference on Theoretical Computer Science*, CEUR Workshop Proceedings, pages 1–17, Rome, Italy, September 2022.

[PMM+13] Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: the full monty. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 217–232. ACM, 2013.

[SF92] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992*, pages 288–298. ACM, 1992.

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 61–78, New York, NY, USA, 1990. Association for Computing Machinery.