CS 131 Compilers: Discussion 9: Scoping, toy type interface and unification

杨易为 季杨彪 尤存翰
{yangyw,jiyb,youch}@shanghaitech.edu.cn

2021 年 5 月 8 日

1 Scope

From today on, we steps into a new topic: semantic. We cast analysis on the grammar from CFG and do as many checks as possible, during which we mark something like type, scope and runtime garbage collector so that they can do better in runtime.

1.1 Dynamic Scoping

A global identifier refers to the identifier associated with the most recent environment, and is uncommon in modern languages. In technical terms, this means that each identifier has a global stack of bindings and the occurrence of a identifier is searched in the most recent binding.

```
/* Since dynamic scoping is very uncommon in the familiar languages, we consider the
 * following pseudo code as our example. It prints 20 in a language that uses dynamic
 * scoping. */
int x = 10;
// Called by g()
int f() {
   return x;
}
// g() has its own variable named as x and calls f()
int g() {
   int x = 20;
   return f();
}
main() {
  printf(g());
}
```

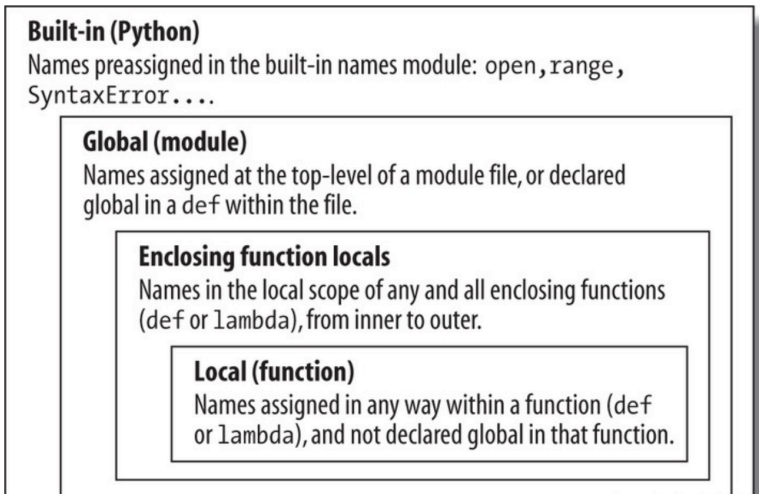
1.2 Lexical Scoping

Scoping refers to the issue of matching identifier declarations with its Uses. The **Scope** of an identifier is the portion of a program where it is accessible.

1. Same identifier may refer to different things in different portions.
2. Different scopes for same identifier **Do Not** overlap.
3. Usually, search for **local** definition first, and if not found, goto its parent Scope.

1.2.1 Python Scoping

In python, we have 4 kinds of scopes and their life cycle, which is called **LEBG**. Notice that every scope maintain its namespace, which is literally a bunch of map of identifiers.



1. **L**ocal scope, that is, the temporary variables defined in the function, when the function ends, the life cycle of the variable ends.
2. **E**nclosed (closure, the local scope of the nested parent function), that is, the local variables of the outer function of the closure, the end of the outer function, the end of the life cycle of the variable.
3. **G**lobal (global variables), that is, variables defined at the module level, the module is destroyed, the life cycle of the variable will end.
4. **B**ulit-in (built-in function) is the python interpreter, the virtual machine built-in variables.

Here we put a benign testcases and adversarial testcases

1. The benign one

```
+--------------------------+
|        +---------------+ |
| def foo|():            | |
|   +----+               | |
|   | x = 1              | |
|   |          +--------+ | |
|   | def inner|():     | | |
|   |   +------+        | | |
|   |   | return x + 1 | | |
|   |   +--------------+ | |
|   | x = 3              | |
|   | print inner()      | |
|   +--------------------+ |
+--------------------------+
```

This inner is a closure(lambda function). The way closures are implemented in Python is to hold a pointer to an external namespace (which can be interpreted as a dictionary).

Question: What's the result of the following output?

```
def make_averager():
    count = 0
    total = 0
    def averager(new_value):
        nonlocal count, total
        count += 1
```

```
        total += new_value
        return total / count
    return averager

averager = make_averager()
print(averager(10))
print(averager(11))
```

Answer:

2. The malicious one

```
def f0(): // NameError: name 'a' has'nt been defined
  a = 123
  def g():
      exec("print(a)")
  g()

def f1(): // print("123\n123")
  a = 123
  def g():
      exec("print(a)")
      print(a)
  g()

def f2(): // NameError: name 'a' has'nt been defined
  a = 123
  def g():
      print(eval("a"))
  g()

def f3(): // print("123\n123")
  a=123
  def g():
      print(eval("a"))
      print(a)
  g()
```

The **eval** is another story. The built-in **eval** function, which evaluates a string as a Python expression(运行字符串中存储的指令). The grammar is listed below.

```
eval(source[, globals[, locals]]) -> value
```

Where **source** is a Python expression or code object return by a function **compile()**, global is dictionary and locals is any mapped object.

1.2.2 Doc is not enough, show me your code

The code is from CPython, a interpreter in C to speed up python.

```
static PyObject *
builtin_eval(PyObject *module, PyObject *const *args, Py_ssize_t nargs)
```

```
{
    PyObject *source;
    PyObject *globals = Py_None;
    PyObject *locals = Py_None;

    source = args[0];
    globals = args[1];
    locals = args[2];
/* ... Checks */
int r = _PyDict_ContainsId(globals, &PyId___builtins__);
if (r == 0) {
    r = _PyDict_SetItemId(globals, &PyId___builtins__,
                          PyEval_GetBuiltins());
}
if (r < 0) {
    return NULL;
}

if (PyCode_Check(source)) {
    if (PySys_Audit("exec", "O", source) < 0) {
        return NULL;
    }

    if (PyCode_GetNumFree((PyCodeObject *)source) > 0) {
        PyErr_SetString(PyExc_TypeError,
            "code object passed to eval() may not contain free variables");
        return NULL;
    }
    return PyEval_EvalCode(source, globals, locals);
}

PyCompilerFlags cf = _PyCompilerFlags_INIT;
cf.cf_flags = PyCF_SOURCE_IS_UTF8;
str = _Py_SourceAsString(source, "eval", "string, bytes or code", &cf, &source_copy);
if (str == NULL)
    return NULL;

while (*str == ' ' || *str == '\t')
    str++;

(void)PyEval_MergeCompilerFlags(&cf);
result = PyRun_StringFlags(str, Py_eval_input, globals, locals, &cf);
Py_XDECREF(source_copy);
return result;
}
```

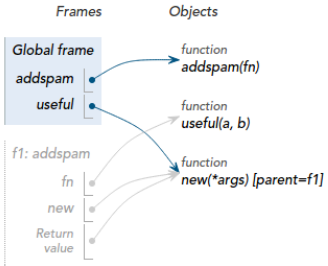
Problem: What is the above code impls of scoping, evaluation at which time and how **eval()** interact with input strings?

Answer:

1.2.3 The Scope of Decorator

When we analyze the calling process of a unknown procedure. The decorator in Python is to make the grammar nice and neat. If you work for Python Open Source community, you must have heard of PEP8. Utilizing Decorator and other grammar sweet is a good way, and also may give you a insight to understand others' reason of writing so. For a general view, a language is all about its grammar sweet.

```
1  def addspam(fn):
2      def new(*args):
3          print "spam,spam,spam"
4          return fn(*args)
5      return new
6
7  @addspam
8  def useful(a,b):
9      print a**2+b**2
10
11 if __name__ == "__main__":
12     useful(2, 3)
```



1.2.4 Make good use of Python Scoping

```
class IntermediateNumber(ABC):
  # class internal namespace, can be shared through the lifetime of the class
  numbers: ClassVar[List[int]]
  digits: int
  def __init__(self, number: Union[List[int], int], digits: int, from_natural: bool):
      self.digits = digits
      if from_natural:
          self.numbers = self.from_natural(number)
      else:
          # builtin function
          if isinstance(number, int): # convert int to List[int]
              numbers = [int(c) for c in str(number)]
          else:
              numbers = number
          # class function
          if not self.check(numbers):
              raise ValueError(f'number not valid: {numbers}')
          self.numbers = numbers
      if digits - 1 > len(self.numbers):
          self.numbers = [0] * (digits - 1 - len(self.numbers)) + self.numbers
  # internal function for stringify
  def __str__(self):
      return f'{self.__class__.__name__[:3]}({"".join(map(str, self.numbers))}){self.custom_str()}'
  # type checking, rhs is callable.
  def check(self, number: List[int]) -> bool:
      return True
  def __add__(self, rhs: IntermediateNumber):
      assert isinstance(self, rhs.__class__)
      return self.add(rhs)
  def __sub__(self, rhs: IntermediateNumber):
      assert isinstance(self, rhs.__class__)
      assert self.to_natural() >= rhs.to_natural()
      return self.sub(rhs)
```

```
  # rhs is callable.
  # generate from natural number
  @abstractmethod
  def from_natural(self, natural: int) -> List[int]:
      pass
  # convert back to natural number
  @abstractmethod
  def to_natural(self) -> int:
      pass
  @abstractmethod
  def add(self, rhs: IntermediateNumber):
      pass
  @abstractmethod
  def sub(self, rhs: IntermediateNumber):
      pass
```

Better for libraries' unit tests and fuzzing tests.

1.2.5 Python interpretor is not as strong as you think

Not every is picklable, a serialization technique when packing tedious python object. We need it because when we need to have multiprocessing or networking stuffs while because of GIL, python is too too too slow. What is not picklable? The lambda functions along with functions and classes defined in the __main__ module, which is the function in the global scope or in a class function.

How to solve it?

Cloud-pickle.

```
from .cloudpickle import (
    _extract_code_globals, _BUILTIN_TYPE_NAMES, DEFAULT_PROTOCOL,
    _find_imported_submodules, _get_cell_contents, _is_importable,
    _builtin_type, _get_or_create_tracker_id,  _make_skeleton_class,
    _make_skeleton_enum, _extract_class_dict, dynamic_subimport, subimport,
    _typevar_reduce, _get_bases, _make_cell, _make_empty_cell, CellType,
    _is_parametrized_type_hint, PYPY, cell_set,
    parametrized_type_hint_getinitargs, _create_parametrized_type_hint,
    builtin_code_type,
    _make_dict_keys, _make_dict_values, _make_dict_items,
)
```

2 Type System

A type is a set of values together with a set of operations on those values. A language's type system specifies which operations are valid for which types.

2.1 Type Inference(not interface in java/ts)

Types of expressions and parameters need not be explicit to have static typing. With the right rules, might infer their types.

The appropriate formalism for type checking is logical rules of inference having the form. cool is a statically typed

```
doh() : Int { (let i : Int <- h in { h <- h + 2; i; } ) };
```

Here the doh can be inferred as the type should be int even doh() does not have a type hint.

2.2 Type Checking Rules

Goal of type checking is to ensure that operations are used with the correct types, enforcing intended interpretation of values.

Notion of "correctness" often depends on what programmer has in mind, rather than what the representation would allow.

Most operations are legal only for values of some types

1. Doesn't make sense to add a function pointer and an integer in C
2. It does make sense to add two integers
3. But both have the same assembly language implementation: movl y, %eax; addl x, %eax

2.3 Type System for a Toy Language

Define a really easy typeof to connect with operational semantic for cool. we only will consider the following subset grammars.

```
defn(I,T,[def(I,T)| _])
defn(I, T, [def(I1,) | R]) :- dif(I, I1), defn(I, T, R).
typeof(X, T, Env) :- defn(X, T, Env).
```

_ means "don't care"or "wildcard", I is defined to have type T if the environment list starts with such a definition, or if I isn't the same as the identifier in the first def but matches the next suitable definition further down the list.

```
e : ID | int
  | [ ((e ,)* e)?  ] /* list */
  | lambda ( ID , e )
  | e ' +'  e
  | e ' <<'  e
  | e ' //'  e
  | cast(e,e)
```

Begin with two typing rules.

```
typeof(X, int,) :- integer(X).
typeof(X, T, Env) :- defn(X, T, Env).
```

1. Write a typing rule such that the type of an empty list is unbounded (e.g [_]).
   Answer:
2. Write a typing rule such that the type of a list is [T], where all list elements are of type T.
   Answer:
3. Write a typing rule such that the type of a lambda is $T1 -> T2$, where $T2$ is the returntype and $T1$ is the type X is bound to within the body of the lambda.
   Answer:
4. Write a typing rule such that the type of $L + R$ isintwhenLandRare both of type int.
   Answer:
5. Write a typing rule such that the type of $L//R$ is [T] whenLandRare both of type [T].
   Answer:
6. Write a typing rule such that the type of $cast(L, R)$ is $T1 -> T2$ when L is of type $T1$ and R is of type $T2$.
   Answer:

Then we can get the reverse opertaion to infer type using typeof:

1. `typeof(f << g, T, [def(f, int->[int]), def(g, int)]).`

   Answer:

2. `typeof(lambda(x, x // x) << [1], T, []).`

   Answer:

3. `typeof(lambda(x, x + x) << 1, T, []).`

   Answer:

4. `typeof(lambda(x, x + x) + 1, T, []).`

   Answer:

5. `typeof(lambda(x, x // x) << [lambda(x, x // x)], T, []).`

   Answer:

6. `typeof(lambda(x,lambda(y, cast(x,y))) << [lambda(x,lambda(y, cast(x,y)))],T, []).`

   Answer:

2.4 Type Unification

To unify two type expressions is to find substitutions for all type variables that make the expressions identical. This technique will be useful for PA3 and PA4, or you will not pass all the testcases. The algorithm that follows treats type expressions as objects (so two type expressions may have identical content and still be differ- ent objects). All type variables with the same name are represented by the same object. It generalizes binding by allowing all type expressions (not just type variables) to be bound to other type expressions.

- For any type expression, define

$$\text{binding}(T) = \begin{cases} \text{binding}(T'), & \text{if } T \text{ is bound to type expression } T' \\ T, & \text{otherwise} \end{cases}$$

- Now proceed recursively:

```
unify (TA,TB):
  TA = binding(TA); TB = binding(TB);
  if TA is TB: return True;  # True if TA and TB are the same object
  if TA is a type variable:
    bind TA to TB; return True
  bind TB to TA;  # Prevents infinite recursion
  if TB is a type variable:
    return True
  # Now check that binding TB to TA was really OK.
  if TA is C(TA1,TA2,...,TAn) and TB is C(TB1,...,TBn):
     return unify(TA1,TB1) and unify(TA2,TB2 and ...
     # where C is some type constructor
  else:  return False
```