ECE326

PROGRAMMING LANGUAGES

Lecture 15 : Reflective Programming

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Introspection

- The ability to examine the type or attribute of a value
  - At runtime
  - Compile-time introspection is called *static introspection*
- Python examples
  - `isinstance(object, cls)`
    - Checks if object is an instance of class

```
class A : pass
class B(A) : pass

>> obj = B()
>> isinstance(obj, B)
True
```

```
>> isinstance(obj, A)
True
>> isinstance(obj, int)
False
>> isinstance(A, B)
False
```

Introspection

- issubclass(cls$_1$, cls$_2$)
  - Checks if class is a subclass of another

```
class A : pass
class B(A) : pass

>> issubclass(B, A)
True
```

```
>> issubclass(A, B)
False
>> obj = B()
>> issubclass(obj, B)
TypeError: obj must be a class
```

- dir(object=None)
  - Returns a list of object’s attributes

```
>> dir(A)
[‘__init__’, ‘__class__’, '__delattr__', '__dict__', …]
```

Introspection

- hasattr(object, name)
  - Checks if string *name* is the name of one of the object’s attribute

```
class A:
  x = 5
  def foo(): pass

>> hasattr(A, ‘x’)
True
```

```
>> hasattr(A, ‘y’)
False
>> my_name = ‘foo’
>> hasattr(A, my_name)
True
```

- type(object)
  - Returns type of object

```
>> a = A()
>> type(a)
<class ‘__main__.A’>
```

```
>> type(A.foo)
<class ‘function’>
>> type(a.foo)
<class ‘method’>
```

Introspection

- C++ Example
  - Runtime Type Information (RTTI)
  - typeid
    - Returns the type id of an object

```
if (typeid(Student) == typeid(*object)) {
    return hash_student(object);
}
```

  - dynamic_cast
    - Downcasts a base class pointer to a subclass pointer, if valid

```
Animal * ap = animals.pop();
Lion * lp = dynamic_cast<Lion *>(ap);
```

Static Introspection

- Introspection at compile time
- Treating compiler as a *white box*
  - The compiler reveals what it knows about an entity
    - type, variable, expression, …etc
  - Make use of how compiler internally represents an entity
- C++ example
  - `decltype`
    - Returns the type of an expression at compile time
    - `typeof` is the non-standard version of decltype

```
decltype(7/2) a = 5;  // a is of type int
```

Reflection

- The ability for a process to introspect and modify itself
  - Changes its own code, such as structure and behavior
  - Can even change the programming language itself
    - E.g. syntax, semantic, implementation
- Process
  - A running instance of a program
- Static reflection
  - Generates compile-time meta-objects
    - E.g. `dir` from Python for C++, only accessible at compile-time

Reification

- Turns abstract representation into concrete data types and/or addressable objects
- Simpler definition
  - Converting compile time types into run-time entities
- Java Example
  - Type information kept to perform runtime type checking

```
String strings[] = {"a", "b"};
Object objects[] = strings;     // allowed at compile time
objects[0] = 5;                 // allowed at compile time

java.lang.ArrayStoreException: java.lang.Integer
```

Type Erasure

- Removal of type information/checks at runtime
  - Type checking at compile-time, none at runtime
- C++ Example

```
struct data {
  int norm;
  int sample[16];
} ;
```

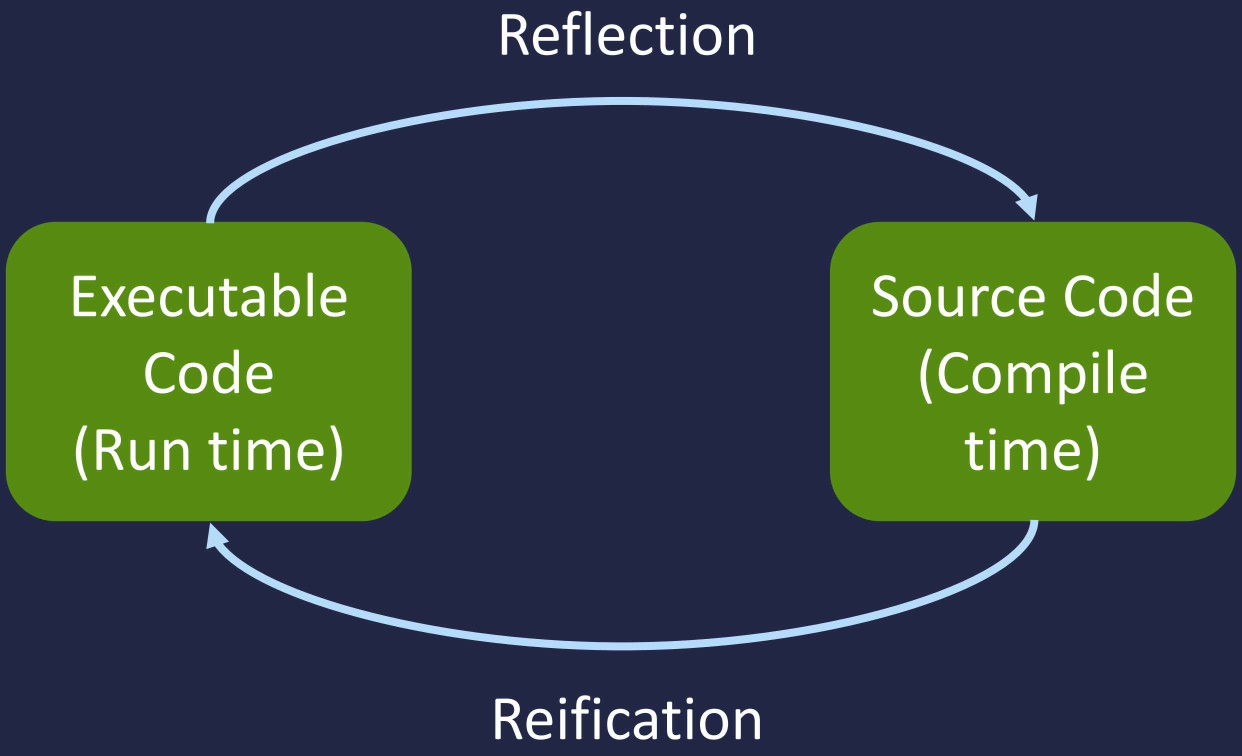
```
int normalized(struct data * d, int i) {
  return d->sample[i] / d->norm;
}
```

```
movslq      %edx, %rdx
movl        4(%rcx,%rdx,4), %eax
cltd
Idivl       (%rcx)
ret
```

Generated code assumes correct structure (struct data) is passed in. No type checking is made at runtime, which improves performance

Reflection and Reification

- Statically-typed, interpreted or byte-compiled languages
  - Anything that uses JVM, E.g. Java, Kotlin



Reflection

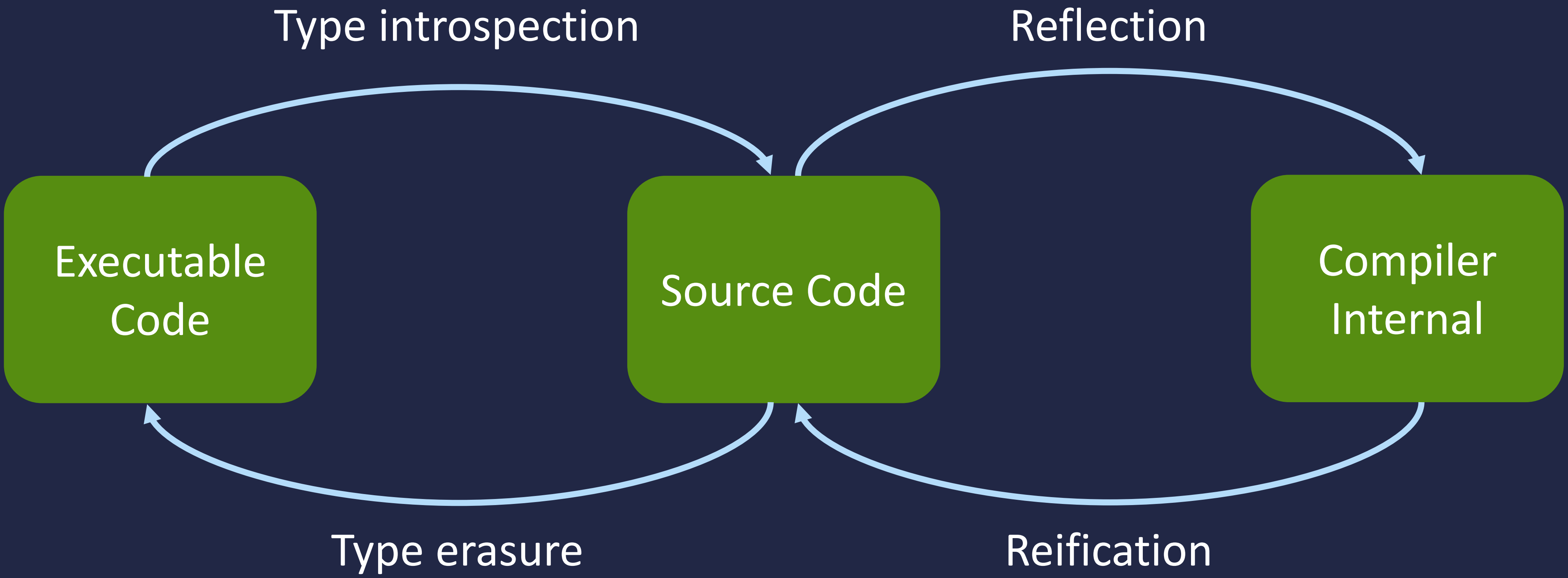
Executable Code (Run time)

Source Code (Compile time)

Reification

Static Reflection

- Statically-typed, systems programming languages



Python Reflection

- `setattr(object, name, value)`
  - Set an object’s attribute with *name* to arbitrary *value*

```
class A:
  x = 5

>> setattr(A, ‘x’, set)
>> A.x
set([])
```

```
>> setattr(A, ‘y’, 7)
>> A.y
7
# equivalent to A.z = None
>> setattr(A, ‘z’, None)
```

- `getattr(object, name, default=None)`
  - Retrieves an attribute by name, return *default* if not found. If default is not specified, raise AttributeError

```
>> getattr(A, ‘y’, 0)
0
```

```
# equivalent to A.y
>> getattr(A, ‘y’)
```

Python Reflection

- `delattr(object, name)`
  - Delete an object’s attribute by *name*

```
class A:
  x = 5

>> delattr(A, ‘x’)
```

```
>> hasattr(A, ‘x’)
False
# equivalent to del A.z
>> delattr(A, ‘z’,)
```

- `globals(), locals(), vars(object)`
  - Returns a dictionary of all global/local/instance variables

```
>> globals()
{'__name__': '__main__', '__doc__': None, '__package__':
None, '__annotations__': {}, '__builtins__': <module
'builtins' (built-in)>, …}
```

Managed Attributes

- Provides control over attribute access
  - E.g. fetch (get), assignment (set), or deletion (del)
- Property
  - Allows attribute access to invoke methods
  - Makes calling methods appear as a data attribute access

```
class Person:
    def get_full(self):
        return self.first + “ ” + self.last

    def set_full(self, value):
        self.first, self.last = value.split(“ ”, maxsplit=1)

    full_name = property(get_full, set_full)
```

Descriptor

- A class that customizes get, set, and/or delete of another object’s attribute
- Similar to property, except more flexible
  - Since it’s a class, it can be subclassed, or inherit another

```
class Descriptor:
    def __get__(self, instance, owner): …
    def __set__(self, instance, value): …
    def __delete__(self, instance): …

class Foo:
    managed = Descriptor()

f = Foo()
f.managed = 5     # calls Descriptor.__set__
```

Descriptor

- `__get__(self, instance, owner)`
  - *instance* is the instance variable, `None` if attribute is accessed through the class (Foo.attr instead of f.attr)
  - *owner* is always the class (e.g. Foo)

```
>> f.managed
# self: Descriptor instance, instance: f, owner: Foo
>> Foo.managed
# self: Descriptor instance, instance: None, owner: Foo
```

- `__set__(self, instance, value)`
  - If not defined, allows attribute to be overwritten!
  - Unlike property, default behaviour makes attribute read-only

Descriptor

```
class CreditCard:
    NUM_DIGITS = 16
    def __init__(self, name, number):
        self.name, self.number = name, number
    class Number:
        def __get__(self, instance, owner):
            return self.number[:-4] + '****'
        def __set__(self, instance, value):
            value = value.replace('-', '')
            if len(value) != instance.NUM_DIGITS:
                raise TypeError('invalid credit card number')
            self.number = value
    number = Number()

card = CreditCard("Jack", "1234-3453-5256-1758")
print(card.number) # prints 123434535256****
```

__setattr__

- Intercepts all assignments to the object’s attribute
- Example

```
class Immutable:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __setattr__(self, name, value):
        raise AttributeError(“cannot update read-only object”)

>> obj = Immutable(5, 6)
>> obj.x = 3
AttributeError: cannot update read-only object
```

__getattr__

- Intercepts all fetch (get) from an object that results in attribute not found
  - Before the AttributeError is raised
- Use case
  - Returning default values on attribute not found
  - Automatic forwarding
- Caveat
  - Does not intercept if method overloads an operator
    - Anything that starts and ends with __ (e.g. __getitem__)

Automatic Forwarding

```
class Hand:
    def __init__(self, cards=tuple()):
        self.cards = list(cards)          # copy the list

    def _points(self):
        return sum(self.cards)
    points = property(_points)

    def __getattr__(self, name):
        return getattr(self.cards, name)

>> p = Hand([2, 3, 4])
>> p.append(9)                  # goes through __getattr__
>> print(p.points)              # points exists – does not go
19                              # through __getattr__
```

__getattribute__

- Intercepts all fetch (get) from an object
  - Also includes those not found (i.e. __getattr__)
- Danger – improper use will result in infinite recursion
  - Use super() instead of self to avoid infinite recursion
- Similar caveat as __getattr__
  - May be bypassed by operator overloading
- Use case
  - Disable access to “private” members

Private Members

```
class Protected:
    def __init__(self, x, y):
        self._x, self._y = x, y

    def getX(self):
        return vars(self)['_x']     # same as self.__dict__[‘_x’]

    def __getattribute__(self, name):
        val = super().__getattribute__(name)
        if name != "__dict__" and name.startswith("_"):
            raise AttributeError(name + “ is a private member")
        return val
```

```
>> p = Protected(5, 7)
>> p._x
AttributeError: _x is a private
member
```

```
>> p.getX()
5
```