Structural Typing and Interfaces

May 7th 2020

Noah Rivkin

Interfaces

- Interfaces are like abstract classes
- They have methods with specific signatures
- A signature describes the name, inputs, and outputs of a function or method
- All of the inputs and outputs have specified types
- The interface acts as a type, so a function can require it as an input

```
// some sort of mangled combination of scala and golang
type Reader : interface{
    def read(n : int) : List[String] // reads n units
}
-> CSVReader
-> IOReader
-> FileReader
-> ....
```

Static vs Dynamic

- Dynamic
  - Duck typing
  - If it quacks like a duck
  - Shares pros and cons of dynamic typing
- Static
  - Structural typing
  - Check before runtime
  - Shares pros and cons of static typing

Advantages and Disadvantages

- Serves as a method of polymorphism/subtyping
- Compatible with inheritance
- Very helpful when implementing a plug-and-play style test harness
- Code made for the interface can accept all objects that implement the interface

- Implementation is (arguably) more complex than inheritance
- Does not allow for super() methods
- Better approximation -> harder to typecheck

Objects

- Any object that implements the interface can be used for in its place
- An object can implement an interface if:
  - It has all of the interface's methods
  - The methods all have the same types as specified in the interface

```
// some sort of mangled combination of scala and golang
class CSVReader : IOReader {
     def read(n : int) : List[String] {
          // reads n lines from the a CSV and parses them
     }
     ....
}
// This implements the IOReader interface
```

Methods

- Function called from an object
- Can reference the object and its attributes
- Has a *this* or *self* parameter
- Can only be called through the object
- There are static methods which are allowed to break these rules, but I am not covering them

Proposed Surface Syntax

```
interface   ::=   :interface interface_id (abstracts)

abstracts   ::=   abstract a_id (params) :type
                  abstracts abstracts

object      ::=   ...
                  class c_id :object (constructor)((methods))
                  class c_id :object (constructor)((methods) implements interface_id)

methods     ::=   method m_id (params) (body) :type
                  methods methods

params      ::=   id :type
                  params param

expr        ::=   ...
                  (def object)
                  (def interface)
                  (new object obj_id (params) (expr))
```

Stack implementation

```
(def :interface stack <T>
    (
        abstract push (s :self i :T) :self
        abstract pop (s: self) :bool
        abstract top (s: self) :T
    )
)
> stack -> env
(def :object int_stack (constructor) :self
    ((
        method push (s :self i :int) (body) :self
        method pop (s :self) (body) :bool
        method top (s :self) (body) :int
    ) implements stack)
)
> int_stack -> env
```

Stack implementation

```
(new int_stack s () (
    do (
        push (s 10)
        push (s 20)
        top (s)
        pop (s)
        top (s)
        pop (s)
        top (s)
        )
))
> true
> true
> 20
> true
> 10
> true
> false
```

TODO

- Multiple interfaces
- Nested interfaces
- Cleaner typing
- Inheritance

Thank You!

Any questions?

Now for the demo…