Recitation 12

Elaine Li (efl9013@nyu.edu)
Office Hours Fridays 1pm – 2pm

Nisarg Patel (nisarg@nyu.edu)
Office Hours Monday 11am – 12pm

Subtyping

- A type defines a set of objects. A subtype (S <: T) defines a set of objects that have at least the methods and fields of T.
- A subtype is a subset of the set defined by its parent type.
  - e.g. Nat <: Int
- If the values of S are a subset of T, then an expression expecting T values can also receive S values.
  - e.g. x+2, f(x), if true then x else y

Function subtyping

- Covariant subtyping: If B <: A, then C -> B <: C -> A.
  - Example: let B be Nat and A be Int, then Nat <: Int
    - Let f1: String -> Nat, f2: String -> Int each interpret the string as a number
    - In a context where I want to interpret a string into a Int, I can use either f1 or f2
    - In a context where I want to interpret a string into an Nat, I can only use f1
    - f1 can be used in all contexts where f2 is expected
    - Therefore f1 <: f2

Function subtyping

- Contravariant subtyping: If B <: A, then B -> C >: A -> C.
  - Example: let B be Nat and A be Int, then Nat <: Int
    - Let f1: Nat -> String, f2: Int -> String each convert the argument as a string
    - In a context where I want to convert a Nat, I can use either f1 or f2
    - In a context where I want to convert an Int, I can only use f2
    - f2 can be used in all contexts where f1 is expected
    - Therefore f2 <: f1

Function subtyping

- “-> type constructor is contravariant in argument type”
  - Given some type S, consider a type constructor C that takes type T1 and returns the type T1 -> S1
  - If T1 <: T2, then T1 -> S >: T2 -> S
  - C(T1) >: C(T2), analogous to Queue[Duck] >: Queue[Bird]
  - Type parameter for C is contravariant

Function subtyping

- “-> type constructor covariant in return type”
  - Given some type S, consider a type constructor C that takes type T1 and returns the type S -> T1
  - If T1 <: T2, then S -> T1 <: S -> T2
  - C(T1) <: C(T2), analogous to Queue[Duck] <: Queue[Bird]
  - Type parameter for C is covariant

Function subtyping

- Arguments ~= contravariant positions, return values ~= covariant positions
- Function type operator is contravariant in argument type and covariant in return type

Example

```
class CoVar[+T](x: T):
  def method1: T = x
  def method2(y: T): List[T] = List(x,y)

class ContraVar[-T](x: T):
  def method1: T = x
  def method2(y: T): List[T] = List(x,y)
```

- At least one method in each class violates the variance annotation of the class' type parameter. Which method?

Example

```
var c := CoVar[Car]
var sc := CoVar[SportsCar]
// SportsCar <: Car

// Because T is covariant
// CoVar[SportsCar] <: CoVar[Car]
// sc must support all methods c supports

c.method2(porsche)
sc.method2(porsche)
c.method2(limo)
sc.method2(limo) // Not allowed!
```

```
class CoVar[+T](x: T):
  def method1: T = x
  def method2(y: T): List[T] = List(x,y)

class ContraVar[-T](x: T):
  def method1: T = x
  def method2(y: T): List[T] = List(x,y)
```

Example

```
var c := ContraVar[Car](limo)
var sc := ContraVar[SportsCar](porsche)
// SportsCar <: Car

// Because T is contravariant
// ContraVar[SportsCar] >: ContraVar[Car]
// c must support all methods sc supports

def race_sportscar(x: sportscar) = {...}
race_sportscar(sc.method1)
race_sportscar(c.method1) // Not allowed!
```

```
class CoVar[+T](x: T):
  def method1: T = x
  def method2(y: T): List[T] = List(x,y)

class ContraVar[-T](x: T):
  def method1: T = x
  def method2(y: T): List[T] = List(x,y)
```