

Covariance and Contravariance

Shou Ya

February 19, 2020

A food processor

假設我們有這樣一個函數：

```
process :: Food -> Something
```

以下哪種調用是正確的？

```
process Apple
```

```
process Food
```

```
process Object
```

當我們講 S 是 T 的子類型 (S is a subtype of T)，符號寫作 $S <: T$ ，意味著在應當使用 T 的地方我們可以安全地使用 S 。對於上面的例子，我們可以說：

`Apple <: Food <: Object`

協變與逆變（Covariance and contravariance）是容器類型的一種屬性。如果說容器類型 $C :: Type \rightarrow Type$ 具有協變屬性，就意味著如果 $A <: B$ ，那麼 $CA <: CB$ 。

我們最熟悉的容器類型一般來說都是協變容器，譬如：

- `Maybe a`: 意味著 `processMaybeFood (Just Apple)` 是合法調用
- `List a`: 意味著 `processListOfFood [Apple, Banana]` 是合法調用

逆變（Contravariance）對我們來講一般來說不那麼熟悉，我們來看下面的函數。

```
filterFood :: (Food -> Bool) -> [Food] -> [Food]
```

`filterFood` 的第一個參數我們可以怎麼傳？下面有三個選項，哪個是合法的？

```
someApplePredicate  :: Apple  -> Bool  
someFoodPredicate   :: Food   -> Bool  
someObjectPredicate :: Object -> Bool
```

Implementing food filter

讓我們來實現一下 `filterFood` 函數吧！

```
filterFood :: (Food -> Bool) -> [Food] -> [Food]
filterFood f []      = []
filterFood f (x:xs) = if f x
                        then x:(filterFood f xs)
                        else  filterFood f xs
```

現在試試將上面的三個斷言（Predicate）傳入，看看哪些不合理？

Which predicates fit and which doesn't?

```
filterFood :: (Food -> Bool) -> [Food] -> [Food]
filterFood f (x:xs) = if f x ...
```

因為我們要對列表中的每個元素調用 **f**，也就是說 **f x** 必須是合法的。

objectPredicate :: Object -> Bool 可以對任何 **Food** 調用
foodPredicate :: Food -> Bool 可以對任何 **Food** 調用
applePredicate :: Apple -> Bool 可能對一些 **Food** 不適用

Subtyping of function types

因此 `filterFood` 可以接受 `objectPredicate` ,
`foodPredicate` 作為第一個參數，但是 `applePredicate` 不行。
換句話說，我們可以說：

$$(\text{Object} \rightarrow \text{Bool}) <: (\text{Food} \rightarrow \text{Bool}) <: (\text{Apple} \rightarrow \text{Bool})$$

剛剛我們講，協變逆變都是 容器的一種屬性。當容器 $C :: Type \rightarrow Type$ 滿足凡是 $A <: B$ ，那麼 $CB <: CA$ ，我們就說容器 C 是具有逆變屬性的。

- 1 上面的函數類型構造器 $(\rightarrow Bool)$ 就是逆變容器。
- 2 這裡的 $Bool$ 類型事實上可以替換成任意類型。
- 3 我們可以換句話來描述，函數類型 $A \rightarrow B$ 在參數 A 上逆變 (Function type $A \rightarrow B$ is contravariant on its argument A .)

Argument type and returning type

很有趣，我們知道了函數類型 $A \rightarrow B$ 在參數 A 上逆變，那麼在返回值 B 上呢？

我們繼續用 `Food` 的例子：

```
readApple  :: (String -> Apple)
```

```
readFood   :: (String -> Food)
```

```
readObject :: (String -> Object)
```

```
getFoodFromFile :: (String -> Food) -> FilePath -> Food
```

```
getFoodFromFile reader path = reader (readFile path)
```

這個例子中，上面三個哪個函數是 `getFoodFromFile` 合法的參數？

```
readApple  :: (String -> Apple)
readFood   :: (String -> Food)
readObject :: (String -> Object)
```

```
getFoodFromFile :: (String -> Food) -> FilePath -> Food
getFoodFromFile reader path = reader (readFile path)
```

我們來分析一下：

- `getFoodFromFile readApple`
- `getFoodFromFile readFood`
- ~~`getFoodFromFile readObject`~~

Covariant on returning value

我們剛剛發現了另一個關於函數類型的事實：

```
(String -> Apple) <: (String -> Food) <: (String -> Object)
```

也就是說，函數類型 $A \rightarrow B$ 在其返回值 B 上協變 (Function type $A \rightarrow B$ is covariant on its returning value B .)

我們剛剛講協變逆變都是容器的屬性。容器的另一個重要屬性就是「容量」。

- `Box a` 的容量是 1
- `Maybe a` 的容量是 1 或者 0
- `List a` 的容量是 0 或更多

協變可以想像為容量為正的容器。換句話說，條件允許的話我們可以從該容器取出值來。

- 給定一個 $(A \rightarrow B)$ ，它可以被看作是一個 `B` 的容器。條件允許（如果我們有一個 `A`）我們可以取出一個 `B`

逆變則相反，逆變容器的容量為負。可以理解為，條件允許我們可以把值放進去。

- 給定一個 $(A \rightarrow B)$ ，它可以被看作是一個 A 的容器。這個容器渴求一個 A 作為輸入，因此我們可以把 A 放進去。

函數的參數和返回值可以被類比為函數的輸入輸出。讀和寫這兩個操作事實上也是輸入輸出。

```
read  :: World -> A
write :: A -> World -> World
```

在上面的讀操作中， A 就是協變的，而在寫操作中， A 就是逆變的。

What's valid in what position?

```
read  :: World -> B
write :: B -> World -> World
```


假如有 $A <: B <: C$ ，那麼

- 如果 B 在一個函數協變位置，那麼換成 C 則是合法的，但換成 A 則不合法
- 如果 B 在一個函數逆變位置，那麼換成 A 則是合法的，但換成 C 則不合法

在這篇博文¹中，王垠提出了這個問題：

```
public static void f() {  
    String[] a = new String[2];  
    Object[] b = a;  
    a[0] = "hi";  
    b[1] = Integer.valueOf(42);  
}
```

這段代碼裡面到底哪一行錯了？為什麼？如果某個 Java 版本能順利運行這段代碼，那麼如何讓這個錯誤暴露得更致命一些？注意這裡所謂的「錯了」是本質上，原理上的，而不一定是 Java 編譯器，IDE 或者運行時報給你的。也就是說，你用的 Java 實現，IDE 都可能是錯的，沒找對真正錯誤的地方，或者沒告訴你真正的原因。

¹<http://www.yinwang.org/blog-cn/2020/02/13/java-type-system> 

```
public static void f() {  
    String[] a = new String[2];  
    Object[] b = a;  
    a[0] = "hi";  
    b[1] = Integer.valueOf(42);  
}
```

現在來試試回答這些問題吧：

- 1 這段代碼裡面到底哪一行錯了？
- 2 為什麼？
- 3 如果某個 **Java** 版本能順利運行這段代碼，那麼如何讓這個錯誤暴露得更致命一些？

- 1 哪行錯了？`b[1] = Integer.valueOf(42)`
- 2 為什麼錯了？我們知道 `String <: Object`，讀取/賦值操作分別等價於下面兩個函數：

```
read0  :: [String] -> String
read0' :: [String] -> Object
```

意味著讀取某個元素無論是通過 `a` 還是 `b` 都是合法的。

```
write1  :: String -> [String] -> [String]
write1' :: Object -> [String] -> [String]
```

意味著寫入某個元素只有通過 `a` 寫才是合法的，而通過 `b` 寫入是非法的。

- 3 如何讓這個錯誤暴露得更致命一些？對於類型 `A`，如果有 `B` 引用同一個值且 `B :> A`，那麼應該禁止 `B` 類型的賦值。

- Invariant: container that are neither covariant or contravariant

```
data F a = F { first :: a, second :: (a -> String) }
```

- Contravariance position on a contravariant position is a covariant position

$$((a \rightarrow x) \rightarrow x) \sim a$$

- Can you express what we learned today with Functor notion?
THEY ARE JUST FUNCTORS!

