Covariance and Contravariance in Scala

Michael Peyton Jones

michaelpj@gmail.com
@mpeytonjones
www.termsandtruthconditions.com

Section 1

Introduction

The WTF

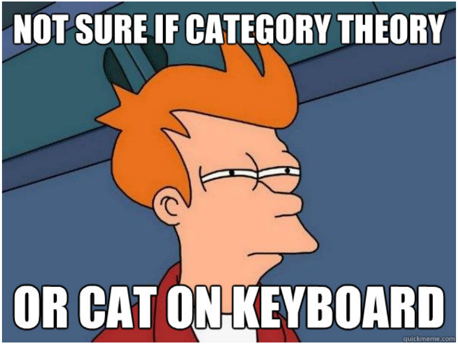
What is going on here?

```
sealed abstract class List[+A] {
    def head : A
    def ::[B >: A](x : B) : List[B] = ...
...
}
```

"Covariant" type parameter. "Contravariant". "Covariant position". Gibberish.

A little bit of category theory

Oh God, category theory.



Section 2

Category Theory

Categories recap

Definition

- Objects
- Morphisms
- Composition
  - Associative
  - Identity

Examples:

- Sets and functions
- Types and functions
- Monoids

The category of types

Category of *types*.

- Objects = types
- Morphisms = functions
- Composition = composition

Functors

Definition

Functor: $F : \mathbf{C} \to \mathbf{D}$

- $F(C)$ is an object in $\mathbf{D}$
- $F(f) : F(C) \to F(D)$ is a morphism in $\mathbf{D}$
- $F(id_C) = id_{F(C)}$
- $F(f \circ g) = F(f) \circ F(g)$

Functors

$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, Rightarrow, "F"] & B \\
FA \arrow[r, "F(f)"'] & FB
\end{tikzcd}
$$

Type Constructors

(Some) type constructors are functors on the category of types.

- e.g. List
- `map(f)` instead of $List(f)$.

Type Constructors

(Some) type constructors are functors on the category of types.

- e.g. List
- `map(f)` instead of $List(f)$.

$$\begin{tikzcd} A \arrow[rr, "f"] \arrow[d, phantom, "\Downarrow"] & & B \\ List[A] \arrow[rr, "map(f)"'] & & List[B] \end{tikzcd}$$

Contravariance

Switches the arrows:

- $f : A \to B$ goes to $F(f) : FB \to FA$ instead of $F(f) : FA \to FB$.
- *Contra*variant functors.
- Covariance is the opposite, i.e. normal.

Contravariance

Switches the arrows:

- $f : A \to B$ goes to $F(f) : FB \to FA$ instead of $F(f) : FA \to FB$.
- *Contra*variant functors.
- Covariance is the opposite, i.e. normal.

$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, phantom, "\Downarrow F"] & B \\
FA & FB \arrow[l, "F(f)"]
\end{tikzcd}
$$

Section 3

Subtyping

Subtyping

Subtyping: class hierarchy.

- "$A < B$ iff A is a subtype of B" is a partial order.
- Which we can look at as a category.
  - Objects = types
  - Morphisms = the existence of a relationship
  - Composition = the relation is transitive

e.g. Sets with inclusion

$$
\begin{tikzcd}
 & \{0,1,2\} & \\
\{0,1\} \arrow[ur] & & \{0,2\} \arrow[ul] \\
 & \{0\} \arrow[ul] \arrow[uu] \arrow[ur] &
\end{tikzcd}
$$

Type constructors again

Type constructors are still (maybe) functors

- Mapped "function" is the subtyping relationship on the new objects.

Type constructors again

Type constructors are still (maybe) functors

- Mapped "function" is the subtyping relationship on the new objects.
- Might go one way, or the other, or none.
  - `List` is covariant, so `Child` $<$ `Parent` implies `List[Child]` $<$ `List[Parent]`
- Covariant, contravariant, invariant.

Section 4

Scala

Type annotations

Scala type annotations for variance:

- $+$ for covariant
- - for contravariant
- nothing for invariant (default)

Type annotations

Scala type annotations for variance:

- $+$ for covariant
- - for contravariant
- nothing for invariant (default)

So

- `Foo[+A]` means `Foo[Child]` $<$ `Foo[Parent]`
- `Bar[-A]` means `Bar[Child]` $>$ `Bar[Parent]`
- `Sock[A]` means no relationship.

Example

```
class GParent
class Parent extends GParent
class Child extends Parent
class Box[+A]
class Box2[-A]

def foo(x : Box[Parent]) : Box[Parent] = identity(x)
def bar(x : Box2[Parent]) : Box2[Parent] = identity(x)

foo(new Box[Child])      // success
foo(new Box[GParent])    // type error

bar(new Box2[Child])     // type error
bar(new Box2[GParent])   // success
```

But...

... what about the really cryptic errors?

```
class Box[+A] {
    def set(x : A) : Box[A]
}
// won't compile
```

But...

... what about the really cryptic errors?

```
class Box[+A] {
    def set(x : A) : Box[A]
}
// won't compile
```

It's all about *functions* (and methods).

Functions

The function trait:

```
trait Function1[-T1, +R] {
    def apply(t : T1) : R
...
}
```

Functions

The function trait:

```
trait Function1[-T1, +R] {
    def apply(t : T1) : R
...
}
```

Weird, huh? We get:

$$\texttt{Function1[GParent, Child]} < \texttt{Function1[Parent, Parent]}$$

Substitution

Why are functions like that?

Substitution

Why are functions like that?

- What are the subtypes of `Function1[A, B]`?

Substitution

Why are functions like that?

- What are the subtypes of `Function1[A, B]`?
- Say `f: Function1[A, B]`, what can we substitute for `f`?

Substitution

Why are functions like that?

- What are the subtypes of `Function1[A, B]`?
- Say `f: Function1[A, B]`, what can we substitute for `f`?
  - Needs to accept a *less* specialized type as input.
  - Can only return a *more* specialized type.

Section 5

Function Functors

Back to Category Theory

Setup:

- For any category **C** we can have the category of the Hom-sets of **C**, **Hom**.
  - Objects = Hom-sets (sets of functions between objects in **C**
  - Morphisms = higher-order functions
  - Composition = composition

Back to Category Theory

Setup:

- For any category **C** we can have the category of the Hom-sets of **C**, **Hom**.
  - Objects = Hom-sets (sets of functions between objects in **C**
  - Morphisms = higher-order functions
  - Composition = composition
- Hom-functor $Hom(-,-) : \mathbf{C}x\mathbf{C} \rightarrow \mathbf{Hom}$, corresponding to the type constructor `Function1[-, -]`.

Back to Category Theory

Setup:

- For any category **C** we can have the category of the Hom-sets of **C**, **Hom**.
  - Objects = Hom-sets (sets of functions between objects in **C**
  - Morphisms = higher-order functions
  - Composition = composition
- Hom-functor $Hom(-,-) : \mathbf{C} x \mathbf{C} \rightarrow \mathbf{Hom}$, corresponding to the type constructor `Function1[-, -]`.
- Claim: $Hom(-,-)$ is contravariant in the first parameter and covariant in the second.
- Actually a bifunctor, let's partially apply.

On the one hand...

Looking at $Hom(A, -)$.

- Takes an object B to the set of functions $A \to B$.
- On functions: given $f : B \to B'$, need a function $Hom(A, f) : Hom(A, B) \to Hom(A, B')$
- $Hom(A, f)(g) = f \circ g$ is the only thing that really works.
- So it's covariant.

On the one hand...

Looking at $Hom(A, -)$.

- Takes an object B to the set of functions $A \to B$.
- On functions: given $f : B \to B'$, need a function $Hom(A, f) : Hom(A, B) \to Hom(A, B')$
- $Hom(A, f)(g) = f \circ g$ is the only thing that really works.
- So it's covariant.

$$
\begin{tikzcd}
B \arrow[rr, "f"] & {} \arrow[d, Rightarrow, "Hom(A, -)"] & B' \\
Hom(A, B) \arrow[rr, "Hom(A, f)"'] & {} & Hom(A, B')
\end{tikzcd}
$$

... on the other hand.

Looking at $Hom(-, B)$.

- Can't really make it covariant.
- $Hom(f, B)(g) = g \circ f$ works, though.
- So it's really contravariant, as g needs to be in $Hom(A', B)$ rather than $Hom(A, B)$.

... on the other hand.

Looking at $Hom(-, B)$.

- Can't really make it covariant.
- $Hom(f, B)(g) = g \circ f$ works, though.
- So it's really contravariant, as g needs to be in $Hom(A', B)$ rather than $Hom(A, B)$.

$$\begin{tikzcd} A \arrow[rr, "f"] & \arrow[d, Rightarrow, "Hom(-, B)"] & A' \\ Hom(A, B) & & Hom(A', B) \arrow[ll, "Hom(f, B)"'] \end{tikzcd}$$

So we've really got a more general result that applies in any category!

Section 6

Back to Earth

Functions and methods

Functions have these properties; shouldn't methods do too?

Functions and methods

Functions have these properties; shouldn't methods do too?

- They do!
- It's not visible in the type system, but it is enforced.

Functions and methods

Functions have these properties; shouldn't methods do too?

- They do!
- It's not visible in the type system, but it is enforced.
- Hence the error, otherwise:
  - replace an instance of `Box[A]` with `Box[B]`
  - so replacing an instance of `Box[A].set(x)` with `Box[B].set(x)`, where `x:B`.
  - can't do this: `set` has to be contravariant in input!

Functions and methods

Functions have these properties; shouldn't methods do too?

- They do!
- It's not visible in the type system, but it is enforced.
- Hence the error, otherwise:
  - replace an instance of `Box[A]` with `Box[B]`
  - so replacing an instance of `Box[A].set(x)` with `Box[B].set(x)`, where `x:B`.
  - can't do this: `set` has to be contravariant in input!
- Likewise if we'd declared `A` to be contravariant, problems with the return type of `set`.
- So `A` has to be invariant.

Aside: Java bashing

Java has covariant arrays: BAD.

```
Integer[] ints = [1,2]
Object[] objs = ints
objs[0] = "I'm an integer!"
```

Compiles, but throws `ArrayStoreException` at runtime.

Alternatives

We don't have to make things invariant: we have type bounds:

```
class BoundedBox[+A] {
    set[B >: A](x : B) : Box[B]
}
```

Bound ensures that the variance requirements are satisfied.

Why bother?

Variance is useful!

- Container types usually want to be covariant.
  - So you can substitute in containers full of subtypes!
  - e.g. List, Stream, etc.
- Types which have an "input" type of some kind usually want to be contravariant.
  - Often this is because there is a function under the hood somewhere.

Summary

- Type constructors may preserve or reverse the subtyping relationship on their input types.
- This is specified by variance annotations.
- Functions have weird variance.
- Methods are (morally) functions!
- Everything is a method!
- Problems? Might need a type bound somewhere.