Contravariance

Intuition building from first principles

Sophie Collard

Disclaimers

Motivation & Audience

Outline

- Prerequisites
- Intuition-building example
- Practical use cases
- Why is contravariance so hard?

Prerequisites

Subtyping

```
enum Animal(val name: String):

  case Cat(override val name: String, livesRemaining: Int)
    extends Animal(name)

  case Dog(override val name: String, breed: DogBreed)
    extends Animal(name)
```

If **Dog** is a subtype of **Animal**

If **Dog** is a subtype of **Animal** (`Dog <: Animal`)

If **Dog** is a subtype of **Animal** (`Dog <: Animal`)

Then whenever an instance of **Animal** is required

If **Dog** is a subtype of **Animal** (`Dog <: Animal`)

Then whenever an instance of **Animal** is required

I may instead provide an instance of **Dog**

If **Dog** is a subtype of **Animal** (`Dog <: Animal`)

Then whenever an instance of **Animal** is required

I may instead provide an instance of **Dog**

```
val a: Animal = Dog(...)
```

Type classes

```scala
trait JsonEncoder[A]:
  def encode(a: A): Json
```

```
trait JsonEncoder[A]:
  def encode(a: A): Json

given JsonEncoder[Animal] with
  def encode(a: Animal): Json =
    ???
```

```
trait JsonEncoder[A]:
  extension (a: A) def encode: Json
```

```
trait JsonEncoder[A]:
  extension (a: A) def encode: Json

given JsonEncoder[Animal] with
  extension (a: Animal) def encode: Json =
    ???
```

Intuition-building example

```
enum Animal(val name: String):

  case Cat(override val name: String, livesRemaining: Int)
    extends Animal(name)

  case Dog(override val name: String, breed: DogBreed)
    extends Animal(name)
```

```
trait Rescue[A]:
  def adopt(name: String): A

trait Clinic[A]:
  def checkup(patient: A): String
```

```scala
trait Rescue[A]:
  def adopt(name: String): A
```

```scala
trait Rescue[A]:
  def adopt(name: String): A

given Rescue[Dog] with
  def adopt(name: String): Dog =
      Dog(name, breed = DogBreed.random())
```

```
trait Rescue[A]:
  def adopt(name: String): A

given Rescue[Animal] with
  def adopt(name: String): Animal =
    Random.between(0, 2) match
      case 0 => Cat(name, livesRemaining = Random.between(0, 7))
      case _ => Dog(name, breed = DogBreed.random())
```

Problem:
I want to adopt an **Animal**

```
def adopt(name: String)(using rescue: Rescue[Animal]): Animal =
  ???
```

```
def adopt(name: String)(using rescue: Rescue[Animal]): Animal =
  rescue.adopt(name)
```

```
def adopt(name: String)(using rescue: Rescue[Animal]): Animal =
  rescue.adopt(name)

val poppy = adopt(name = “Poppy”)
```

Problem:
I want to adopt an **Animal**
But there are only **Dog** rescues near me

```
def adopt(name: String)(using rescue: Rescue[Animal]): Animal =
  rescue.adopt(name)

// Will this compile?
val poppy = adopt(name = “Poppy”)
```

```
def adopt(name: String)(using rescue: Rescue[Animal]): Animal =
  rescue.adopt(name)

// Will this compile?
val poppy = adopt(name = “Poppy”) ❌
```

```
No given instance of type Rescue[Animal] was found for
parameter rescue of method adopt in object Main
```

I want to adopt an **Animal**

I want to adopt an **Animal**

Dog is a subtype of **Animal**

I want to adopt an **Animal**

Dog is a subtype of **Animal**

Can I adopt an **Animal** from a **Dog** rescue?

I want to adopt an **Animal**

Dog is a subtype of **Animal**

Can I adopt an **Animal** from a **Dog** rescue?

Yes

I want to adopt a **Dog**

I want to adopt a **Dog**

Dog is a subtype of **Animal**

I want to adopt a **Dog**

Dog is a subtype of **Animal**

Can I adopt a **Dog** from an **Animal** rescue?

I want to adopt a **Dog**

Dog is a subtype of **Animal**

Can I adopt a **Dog** from an **Animal** rescue?

No

```
def adopt(name: String)(using rescue: Rescue[Animal]): Animal =
  rescue.adopt(name)

// Will this compile?
val poppy = adopt(name = “Poppy”)(using summon[Rescue[Dog]])
```

```
def adopt(name: String)(using rescue: Rescue[Animal]): Animal =
  rescue.adopt(name)

// Will this compile?
val poppy = adopt(name = “Poppy”)(using summon[Rescue[Dog]]) ❌
```

```
Found:  Rescue.given_Rescue_Dog.type
Required: Rescue[Animal]
```

```
Found:  Rescue[Dog]
Required: Rescue[Animal]
```

```
type mismatch;
 found   : Rescue[Dog]
 required: Rescue[Animal]

Note: Dog <: Animal, but trait Rescue is invariant in type A.

You may wish to define A as +A instead. (SLS 4.5)
```

I want to adopt an **Animal**

Dog is a subtype of **Animal**

Can I adopt an **Animal** from a **Dog** rescue?

Yes

I want to adopt an **Animal**

Dog is a subtype of **Animal** (`Dog <: Animal`)

Can I adopt an **Animal** from a **Dog** rescue?

Yes (`if Rescue[Dog] <: Rescue[Animal]`)

```
trait Rescue[+A]:
  def adopt(name: String): A
```

```
trait Rescue[+A]:
  def adopt(name: String): A

// Rescue is covariant in A
// If Dog <: Animal
// Then Rescue[Dog] <: Rescue[Animal]
```

```scala
def adopt(name: String)(using rescue: Rescue[Animal]): Animal =
  rescue.adopt(name)

val poppy = adopt(name = “Poppy”)(using summon[Rescue[Dog]])
```

```
def adopt(name: String)(using rescue: Rescue[Animal]): Animal =
  rescue.adopt(name)

val poppy = adopt(name = “Poppy”)
```

That was **covariance**

Invariance

```
trait Rescue[A] { ... }


// Rescue is invariant in A
// Even though Dog <: Animal
// There is no relation between Rescue[Dog] and Rescue[Animal]
```

Covariance

```
trait Rescue[+A] { ... }


// Rescue is covariant in A
// If Dog <: Animal
// Then Rescue[Dog] <: Rescue[Animal]
```

```
trait Clinic[A]:
  def examine(patient: A): String
```

```
trait Clinic[A]:
  def examine(patient: A): String

given Clinic[Dog] with
  def examine(dog: Dog): String =
    s”${dog.name} is a dog of breed ${dog.breed}”
```

```
trait Clinic[A]:
  def examine(patient: A): String

given Clinic[Animal] with
  def examine(patient: Animal): String =
    patient match
      case Cat(name, lr) => s”$name is a cat with $lr lives remaining”
      case Dog(name, breed) => s”$name is a dog of breed $breed”
```

Problem:

I want to take my **Dog** for a checkup

```
def examine(dog: Dog)(using clinic: Clinic[Dog]): String =
  ???
```

```
def examine(dog: Dog)(using clinic: Clinic[Dog]): String =
  clinic.examine(dog)
```

```
def examine(dog: Dog)(using clinic: Clinic[Dog]): String =
  clinic.examine(dog)

val médor = Dog(“Médor”, breed = DogBreed.Labrador)

val examinationReport = examine(médor)
```

Problem:

I want to take my **Dog** for a checkup

But there are only **Animal** clinics in my area

```scala
def examine(dog: Dog)(using clinic: Clinic[Dog]): String =
  clinic.examine(dog)

val médor = Dog(“Médor”, breed = DogBreed.Labrador)

// Will this compile?
val examinationReport = examine(médor)
```

```
def examine(dog: Dog)(using clinic: Clinic[Dog]): String =
  clinic.examine(dog)

val médor = Dog(“Médor”, breed = DogBreed.Labrador)

// Will this compile?
val examinationReport = examine(médor) ❌
```

```
No given instance of type Clinic[Dog] was found for parameter
rescue of method adopt in object Main
```

I want to take my **Dog** for a checkup

I want to take my **Dog** for a checkup

Dog is a subtype of **Animal**

I want to take my **Dog** for a checkup

Dog is a subtype of **Animal**

Can I take my **Dog** to an **Animal** clinic?

I want to take my **Dog** for a checkup

Dog is a subtype of **Animal**

Can I take my **Dog** to an **Animal** clinic?

Yes

I want to take my **Animals** for a checkup

I want to take my **Animals** for a checkup

Dog and **Cat** are subtypes of **Animal**

I want to take my **Animals** for a checkup

Dog and **Cat** are subtypes of **Animal**

Can I take my **Animals** to a **Dog** clinic?

I want to take my **Animals** for a checkup

Dog and **Cat** are subtypes of **Animal**

Can I take my **Animals** to a **Dog** clinic?

No

```
def examine(dog: Dog)(using clinic: Clinic[Dog]): String =
  clinic.examine(dog)

val médor = Dog(“Médor”, breed = DogBreed.Labrador)

// Will this compile?
val examinationReport = examine(médor)(using summon[Clinic[Animal]])
```

```
def examine(dog: Dog)(using clinic: Clinic[Dog]): String =
  clinic.examine(dog)

val médor = Dog(“Médor”, breed = DogBreed.Labrador)

// Will this compile?
val examinationReport = examine(médor)(using summon[Clinic[Animal]]) ❌
```

```
Found:  Clinic.given_Clinic_Animal.type
Required: Clinic[Dog]
```

```
Found:  Clinic[Animal]
Required: Clinic[Dog]
```

```
type mismatch;
 found   : Clinic[Animal]
 required: Clinic[Dog]

Note: Dog <: Animal, but trait Clinic is invariant in type A.

You may wish to define A as -A instead. (SLS 4.5)
```

I want to take my **Dog** for a checkup

Dog is a subtype of **Animal**

Can I take my **Dog** to an **Animal** clinic?

Yes

I want to take my **Dog** for a checkup

Dog is a subtype of **Animal** (`Dog <: Animal`)

Can I take my **Dog** to an **Animal** clinic?

Yes (`if Clinic[Animal] <: Clinic[Dog]`)

```scala
trait Clinic[-A]:
  def examine(patient: A): String
```

```scala
trait Clinic[-A]:
  def examine(patient: A): String

// Clinic is contravariant in A
// If Dog <: Animal
// Then Clinic[Animal] <: Clinic[Dog]
```

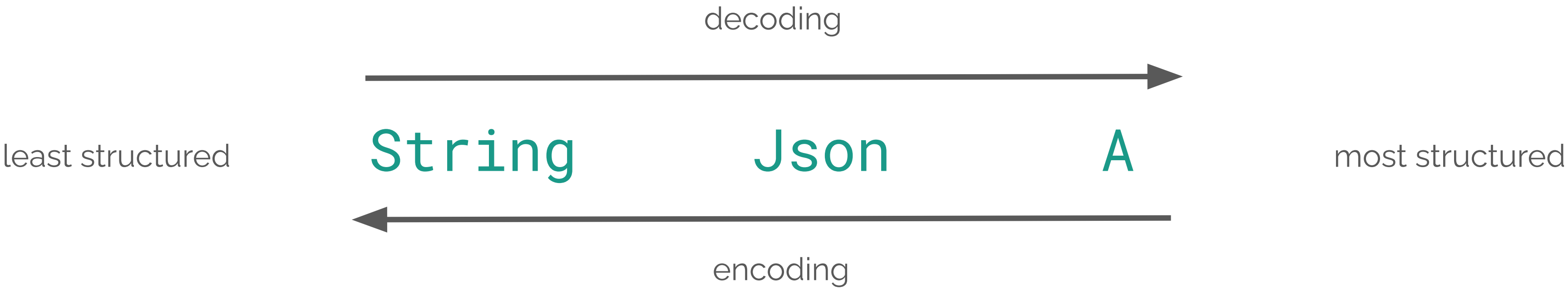
Contravariance

```
trait Clinic[-A] { ... }


// Clinic is contravariant in A
// If Dog <: Animal
// Then Clinic[Animal] <: Clinic[Dog]
```

Contravariance in practice

Codecs

```
trait JsonDecoder[+A]:
  extension (json: Json) def decode: Either[DecodeError, A]

trait JsonEncoder[-A]:
  extension (a: A) def encode: Json
```



decoding

least structured

`String` `Json` `A`

most structured

encoding

```scala
trait JsonDecoder[+A]:
  extension (json: Json) def decode: Either[DecodeError, A]

object JsonDecoder:
  // summoner
  def apply[A](using jd: JsonDecoder[A]): JsonDecoder[A] = jd
```

```scala
trait JsonEncoder[-A]:
  extension (a: A) def encode: Json

object JsonEncoder:
  // summoner
  def apply[A](using je: JsonEncoder[A]): JsonEncoder[A] = je
def encodeDog(dog: Dog, encoder: JsonEncoder[Dog]): Json =
  encoder.encode(dog)
```

```
def encodeDog(dog: Dog, encoder: JsonEncoder[Dog]): Json =
  encoder.encode(dog)

val médor = Dog(name = “Médor”, breed = DogBreed.Labrador)

encodeDog(médor, JsonEncoder[Dog]) // Compiles
```

```
def encodeDog(dog: Dog, encoder: JsonEncoder[Dog]): Json =
  encoder.encode(dog)

val médor = Dog(name = “Médor”, breed = DogBreed.Labrador)

encodeDog(médor, JsonEncoder[Animal]) // Also compiles
```

```scala
trait JsonDecoder[+A]:
  extension (json: Json) def decode: Either[DecodeError, A]

trait JsonEncoder[-A]:
  extension (a: A) def encode: Json
```

Function inputs

```
trait Function1[-T1, +R]:
  def apply(v1: T1): R
```

```
trait Function2[-T1, -T2, +R]:
  def apply(v1: T1, v2: T2): R
```

```
// This works thanks to the input type parameter being contravariant
// i.e. thanks to Function1[Animal, String] <: Function1[Dog, String]

val funcOnAnimal: Function1[Animal, String] = ???

val funcOnDog: Function1[Dog, String] = funcOnAnimal
```

```scala
// This works thanks to the output type parameter being covariant
// i.e. thanks to Function1[String, Dog] <: Function1[String, Animal]

val funcForDog: Function1[String, Dog] = ???

val funcForAnimal: Function1[String, Animal] = funcForDog
```

So, when do we use co- and contravariance?

```scala
trait Rescue[+A]:
  def adopt(name: String): A

trait JsonDecoder[+A]:
  def decode(json: Json): Either[DecodeError, A]

trait Function1[-T1, +R]:
  def apply(v1: T1): R
```

```scala
trait Clinic[-A]:
  def examine(patient: A): String

trait JsonEncoder[-A]:
  def encode(a: A): Json

trait Function1[-T1, +R]:
  def apply(v1: T1): R
```

Why is contravariance so hard?

It is far less common than **covariance**

I suspect people sometimes attribute to **covariance** behaviour which really is provided by **subtyping**

I suspect people sometimes attribute to **covariance** behaviour which really is provided by **subtyping**

This does not work with **contravariance**

```
trait Func[+R]:
  def apply(): R
```

```
trait Func[+R]:
  def apply(): R

val f: Func[Animal] = () => Dog(“Médor”, DogBreed.Labrador)
```

```
trait Func[+R]:
  def apply(): R

// This works thanks to subtyping (Dog <: Animal)
val f: Func[Animal] = () => Dog(“Médor”, DogBreed.Labrador)
```

```
trait Func[R]:
  def apply(): R

// It still works if you make Func invariant in R
val f: Func[Animal] = () => Dog(“Médor”, DogBreed.Labrador)
```

```
trait Func[-R]:
  def apply(): R

// This will never work, despite Func being contravariant in R
val f: Func[Dog] = () => {
  val animal: Animal = ???
  animal
}
```

```
trait Func[+R]:
  def apply(): R

// This works thanks to subtyping (Dog <: Animal)
val f: Func[Animal] = () => Dog(“Médor”, DogBreed.Labrador)
```

```
trait Func[+R]:
  def apply(): R

val f: Func[Dog] = () => Dog(“Médor”, DogBreed.Labrador)

// This works thanks to covariance (Func[Dog] <: Func[Animal])
val hof: Func[Func[Animal]] = () => f
```

```
trait Func[R]:
  def apply(): R

val f: Func[Dog] = () => Dog(“Médor”, DogBreed.Labrador)

// In fact, it’ll stop compiling if you make Func invariant in R
val hof: Func[Func[Animal]] = () => f
```

Recap

Invariance

```
trait JsonDecoder[A] { ... }


// JsonDecoder is invariant in A
// Even though Dog <: Animal
// JsonDecoder[Dog] and JsonDecoder[Animal] are not related
```

Covariance

```
trait JsonDecoder[+A] { ... }


// JsonDecoder is covariant in A
// If Dog <: Animal
// Then JsonDecoder[Dog] <: JsonDecoder[Animal]
```

Covariance

```
// Used for output type parameters

trait JsonDecoder[+A]:
  extension (json: Json) def decode: Either[DecodeError, A]

trait Function1[-T1, +R]:
  def apply(v1: T1): R
```

Contravariance

```
trait JsonEncoder[-A] { ... }


// JsonEncoder is contravariant in A
// If Dog <: Animal
// Then JsonEncoder[Animal] <: JsonEncoder[Dog]
```

Contravariance

```
// Used for input type parameters

trait JsonEncoder[-A]:
  extension (a: A) def encode: Json

trait Function1[-T1, +R]:
  def apply(v1: T1): R
```

Thank you!