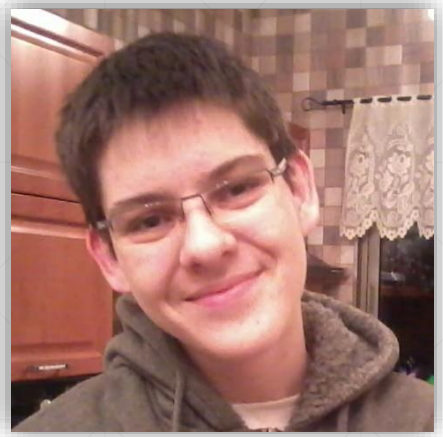
Optics with Monocle

Modeling the part and the whole

By Ilan Godik (@IlanGodik on Twitter, @NightRa on IRC)

About me

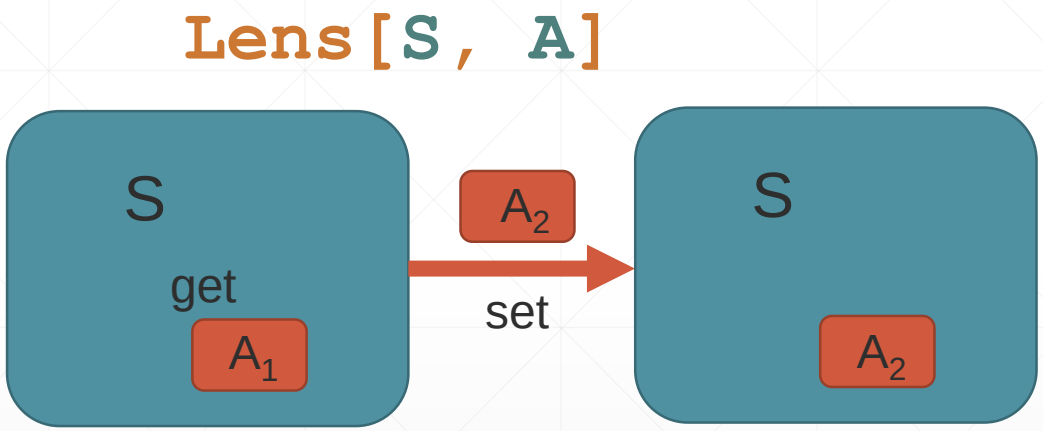


- A CS Undergraduate at Haifa University
- A contributor to the Monocle library
- Functional programming lover.

What are lenses?

In the simplest model, it's just a pair of a getter and a setter:

```
trait Lens[S,A]{
  def get(s: S): A
  def set(a: A)(s: S): S
}
```



Classic lens example

Updating nested structures is verbose and painful

```
case class Person(fullName: String, address: Address)
case class Address(city: String, street: Street)
case class Street(name: String, number: Int)
```

```
person.copy(
  address = person.address.copy(
    street = person.address.street.copy(
      name = person.address.street.name.capitalize
    )
  )
)
```

Monocle Lenses

Define lenses once, and compose as you wish

```
case class Person(fullName: String, address: Address)
case class Address(city: String, street: Street)
case class Street(name: String, number: Int)

val address = Lenser[Person](_.address)
val street  = Lenser[Address](_.street)
val name    = Lenser[Street](_.name)
```

```
(address composeLens street composeLens name).modify(_.capitalize)(person)
```

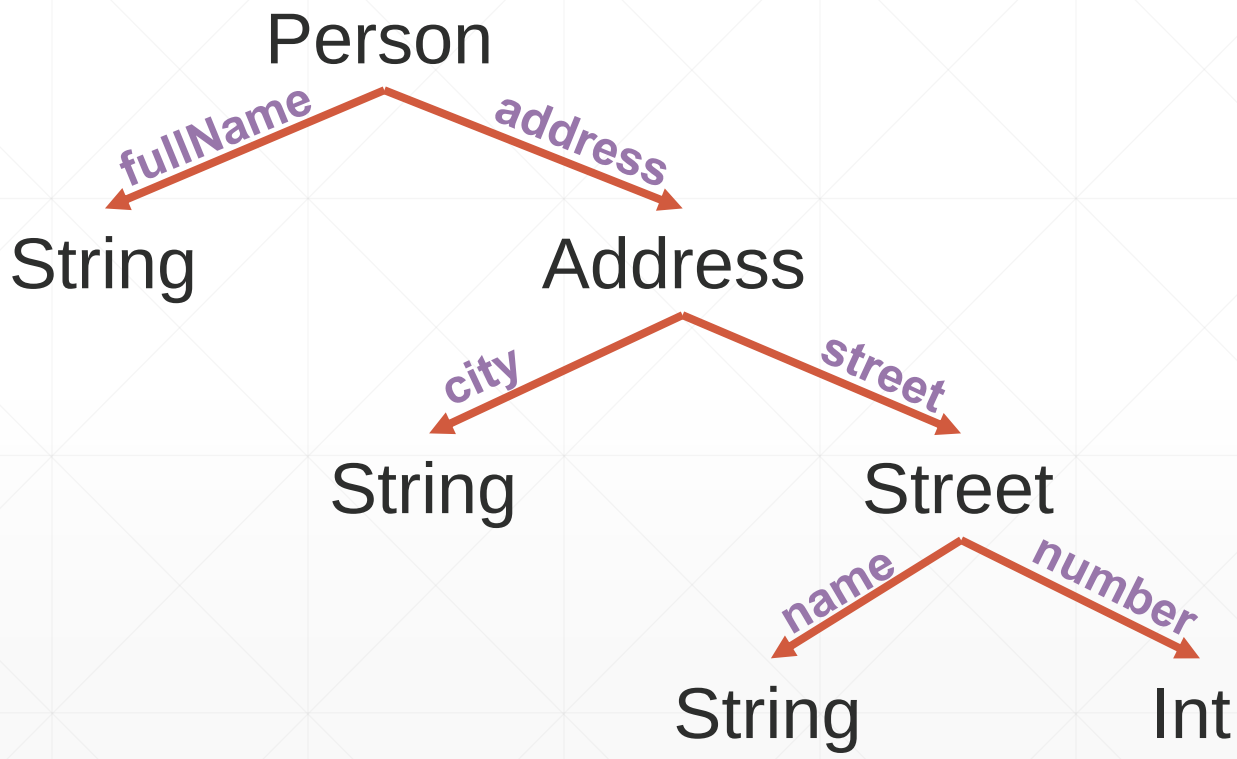
Lenses Macro annotation

Awesome, now also IDE friendly (Intellij support)

```
@Lenses case class Person(fullName: String, address: Address)
@Lenses case class Address(city: String, street: Street)
@Lenses case class Street(name: String, number: Int)
import Person._, Address._, Street._
```

```
(address composeLens street composeLens name).modify(_.capitalize)(person)
```

The object graph

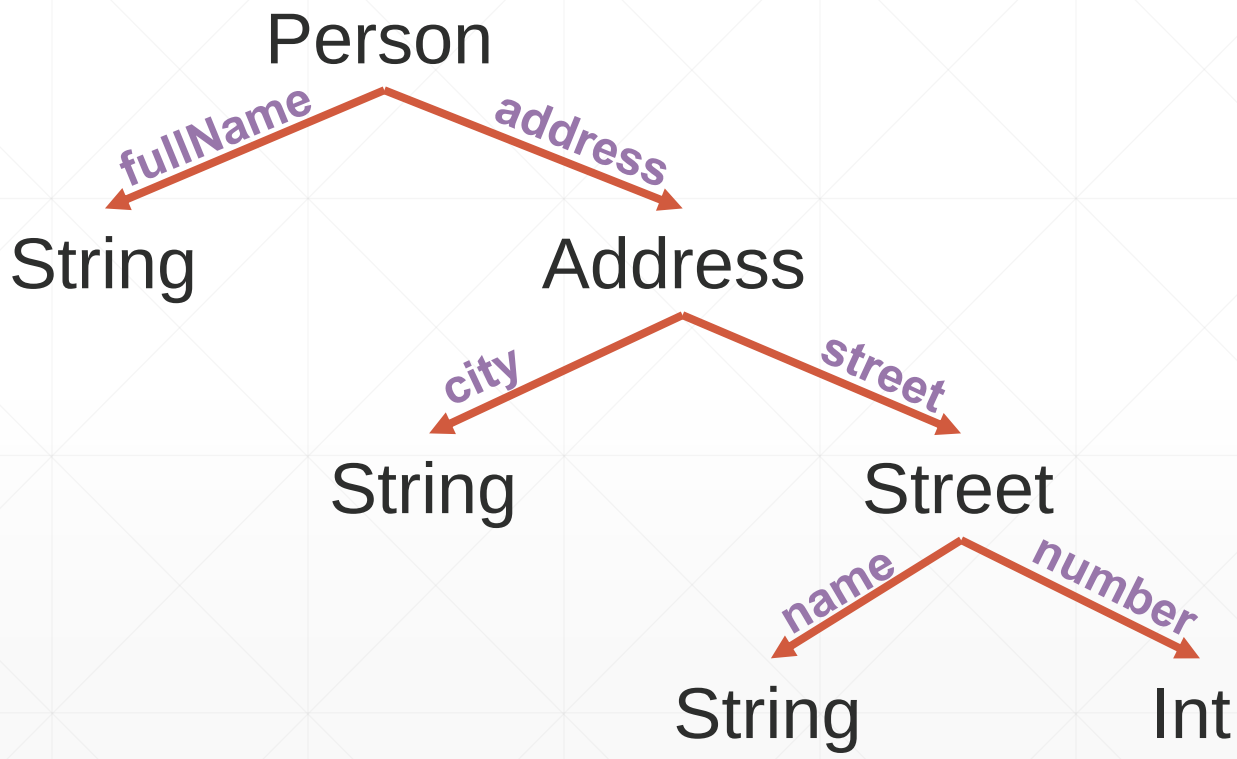


Code size growth

- Vanilla Scala: $O(h^2)$
- Lenses: $O(h)$

  Where h is the height of the object graph

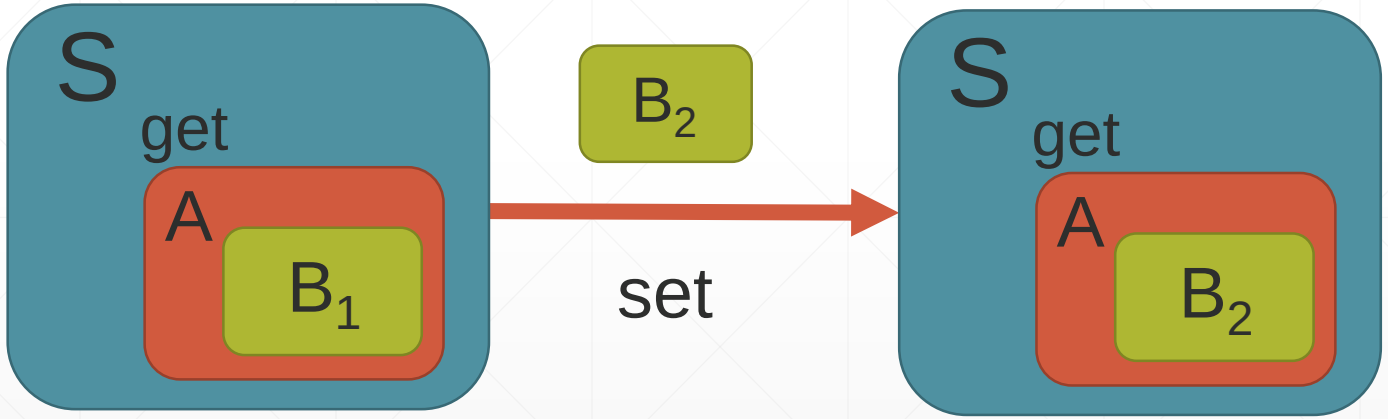
Composition: Follow the arrows



Lens composition

We compose lenses exactly how we would do it with copy.

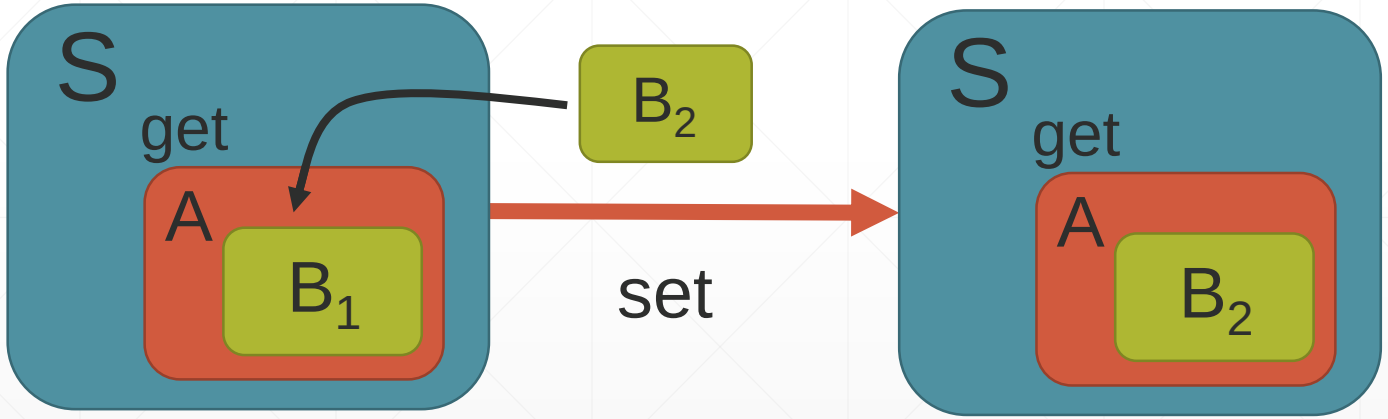
```
Lens[S,A] compose Lens[A,B]
```



Lens composition

We compose lenses exactly how we would do it with copy.

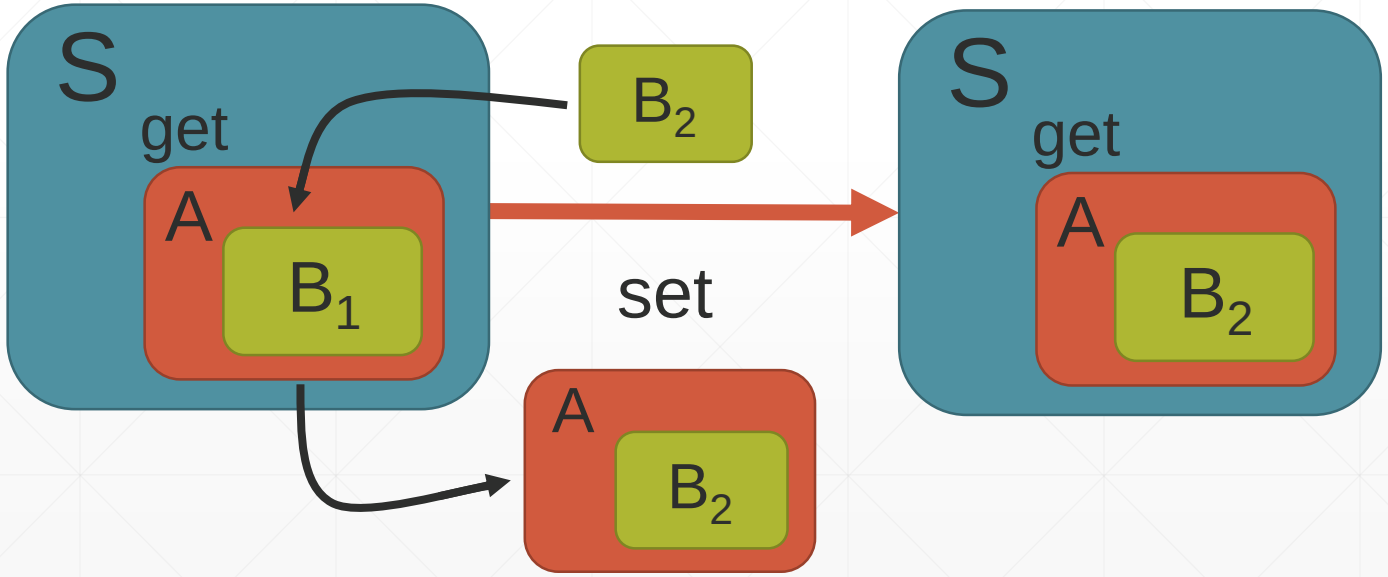
```
Lens[S,A] compose Lens[A,B]
```



Lens composition

We compose lenses exactly how we would do it with copy.

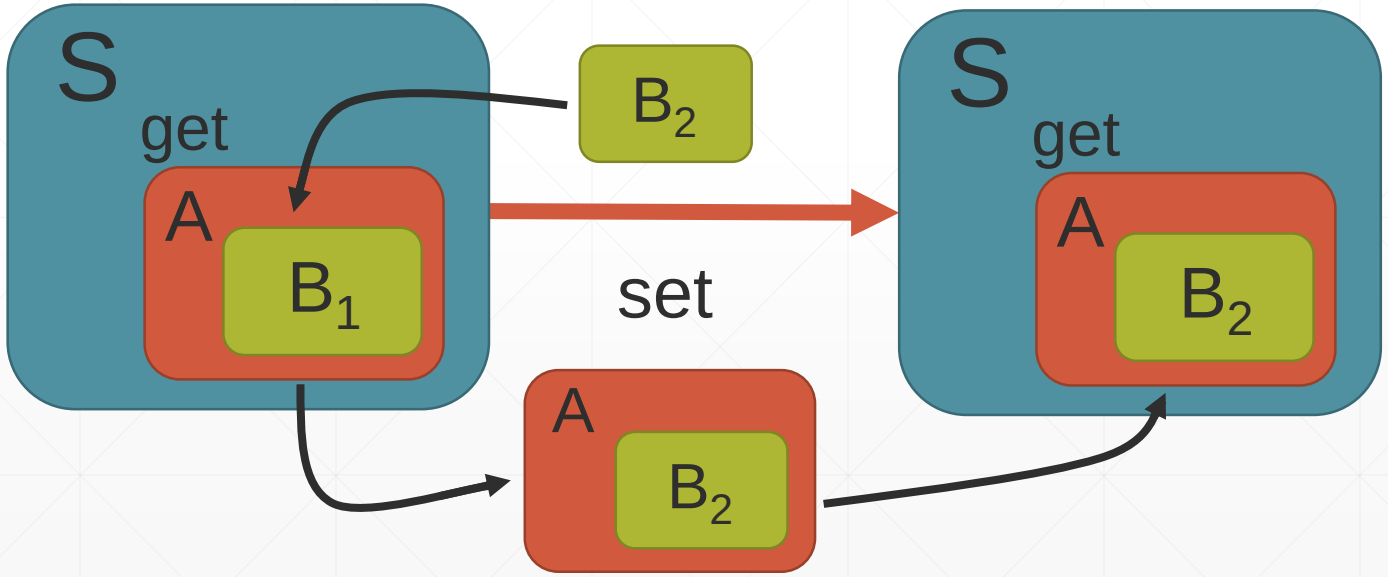
```
Lens[S,A] compose Lens[A,B]
```



Lens composition

We compose lenses exactly how we would do it with copy.

```
Lens[S,A] compose Lens[A,B]
```



Polymorphic Lenses

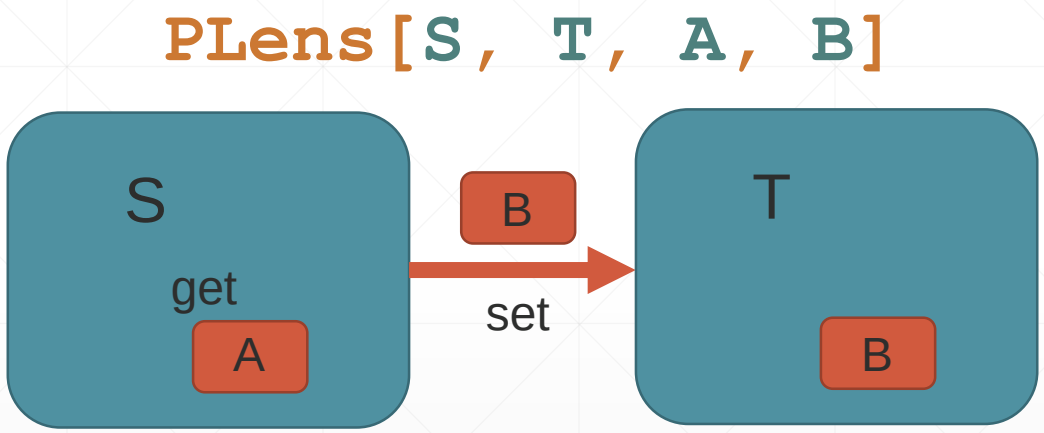
Can we change the type of part of the structure while setting?

```
first.set("Hello")((1,2)) == ("Hello",2)
```

Polymorphic Lenses

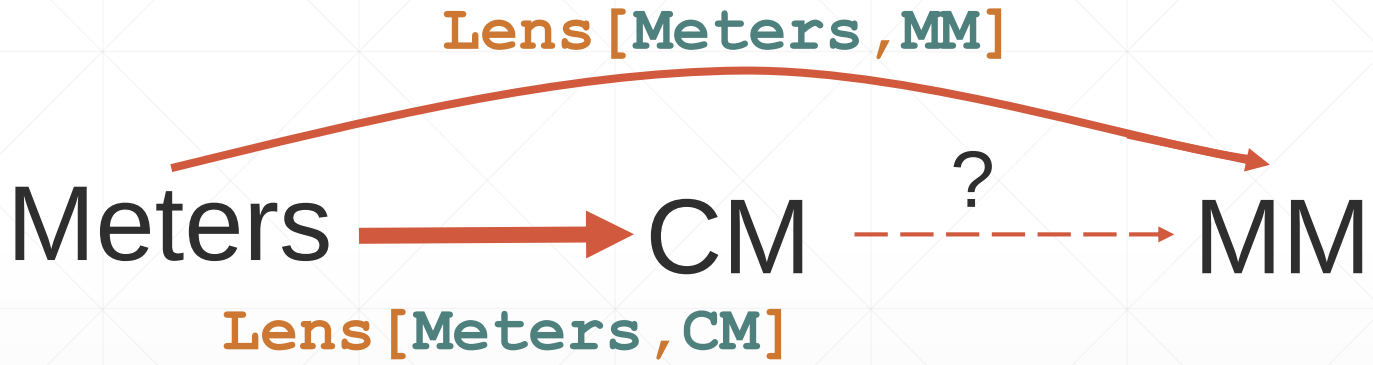
Can we change the type of part of the structure while setting?

```
trait PLens[S, T, A, B] {
  def get(s: S): A
  def set(b: B)(s: S): T
}
```



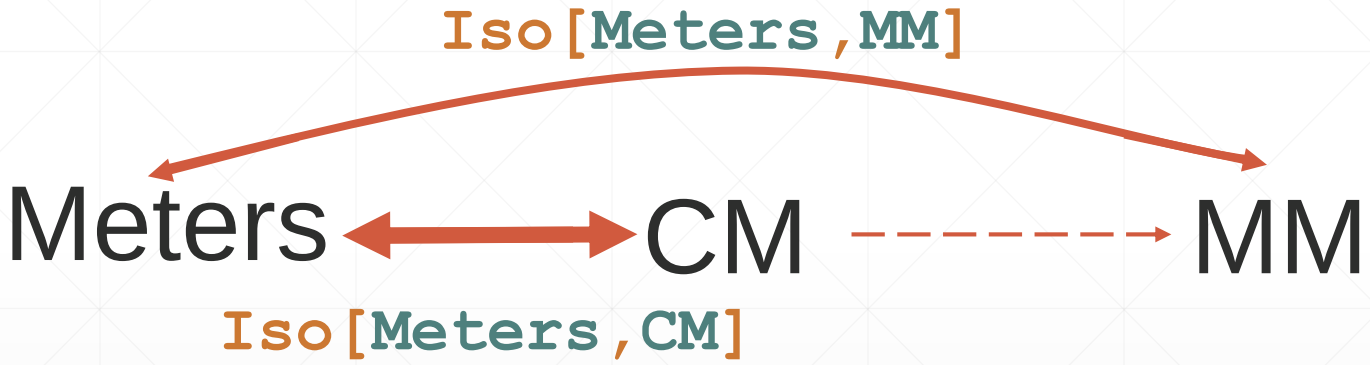
Example: Unit conversion

Optics as different points of view of data



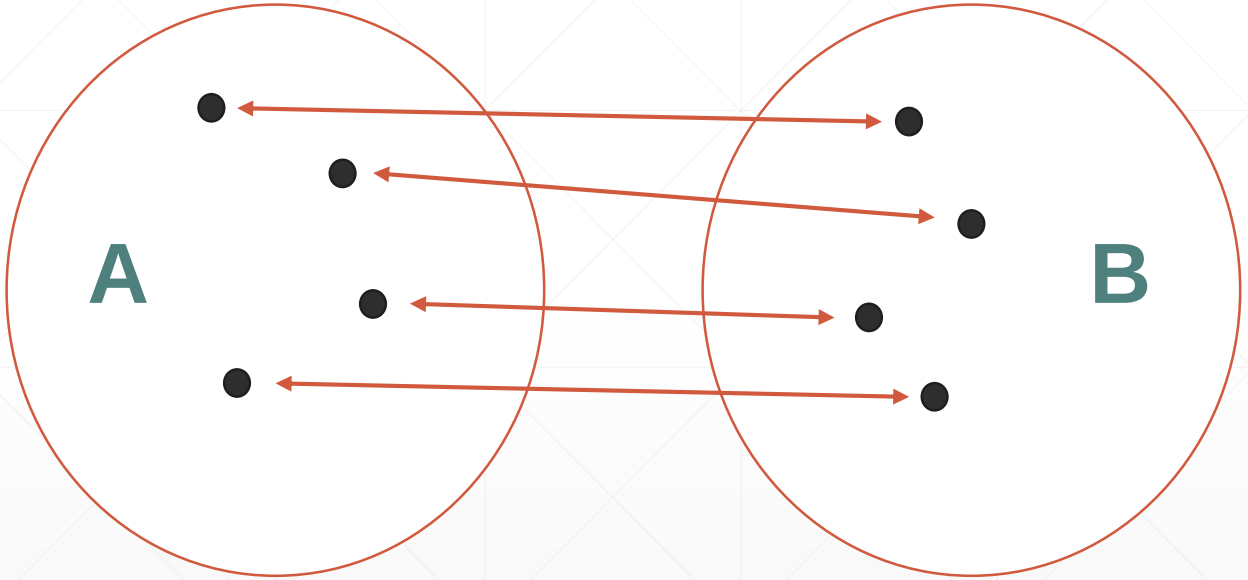
Isomorphisms

Do we get more power if our lens is bidirectional?



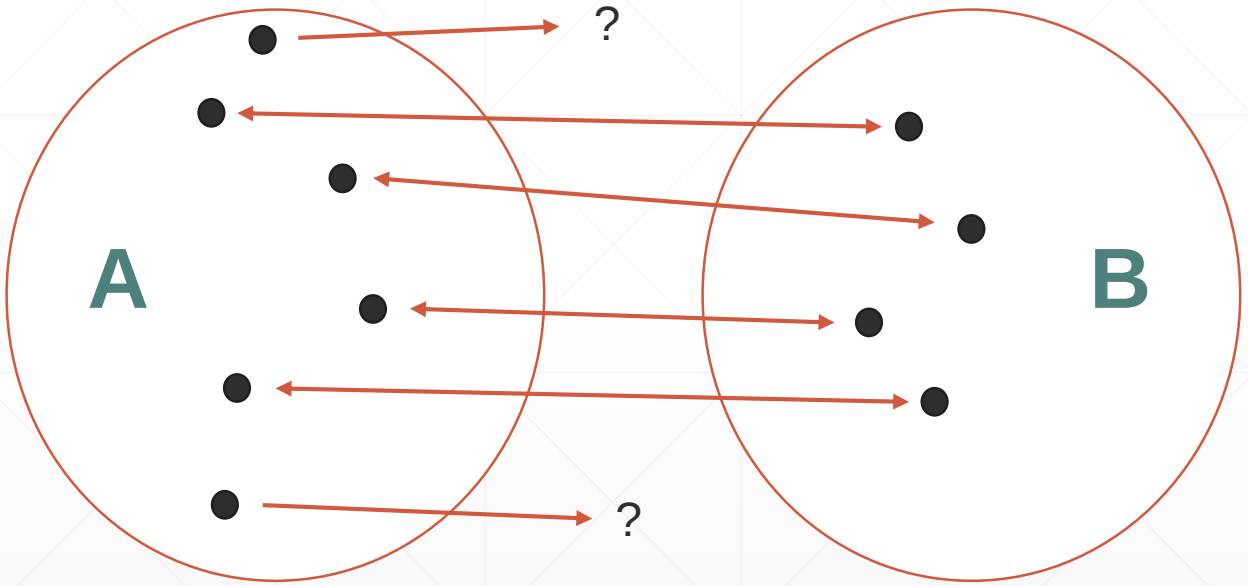
Isomorphisms

Isomorphisms as a bijection: A function with an inverse



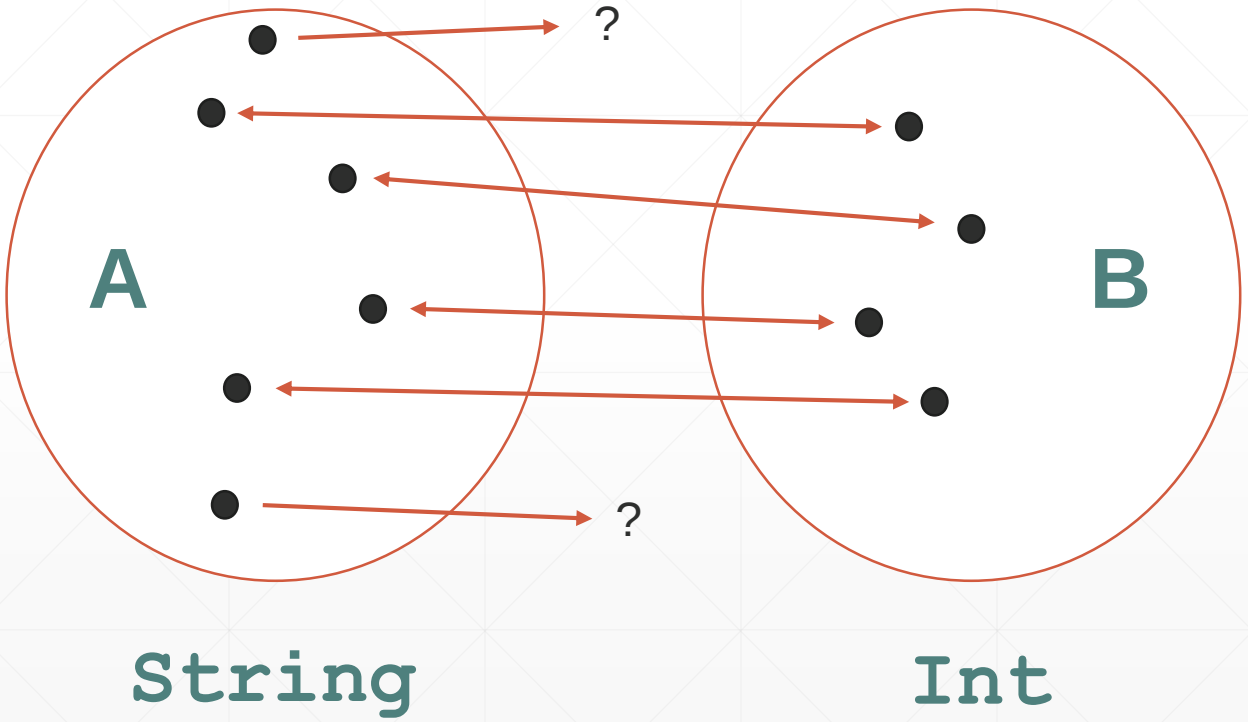
Prisms

What if we don’t have such a nice correspondence?



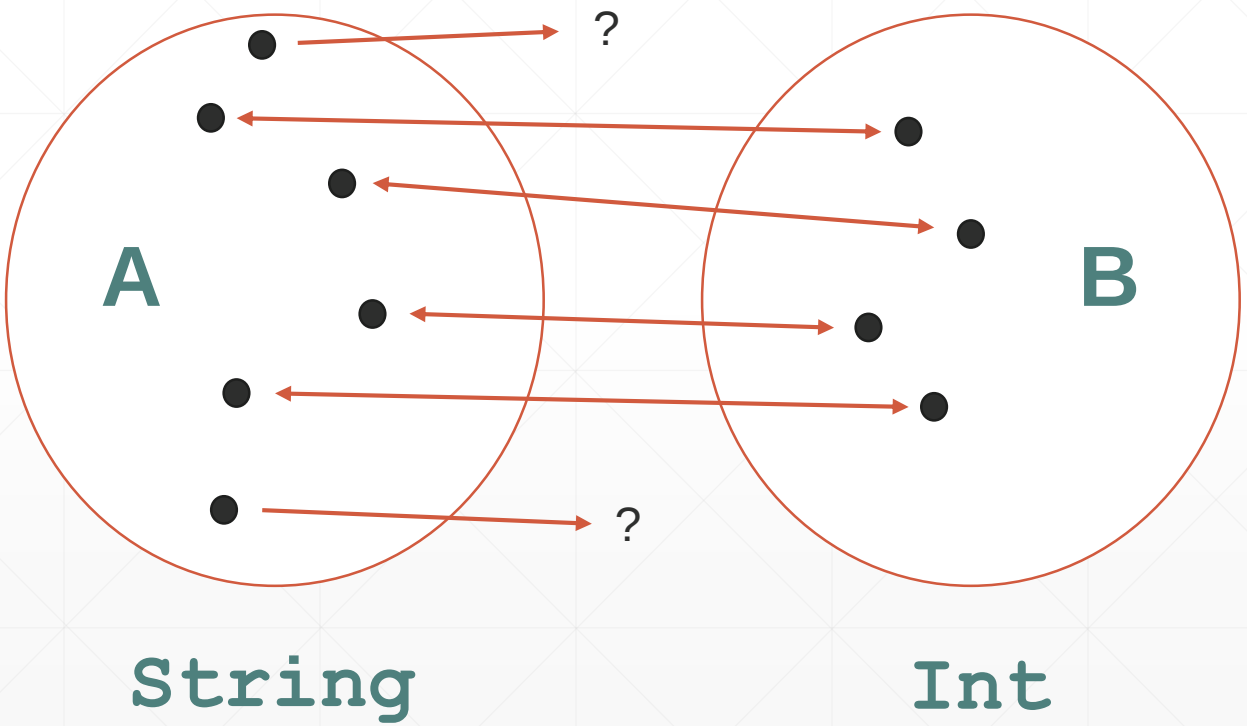
Prisms

What if we don’t have such a nice correspondence?



Prisms

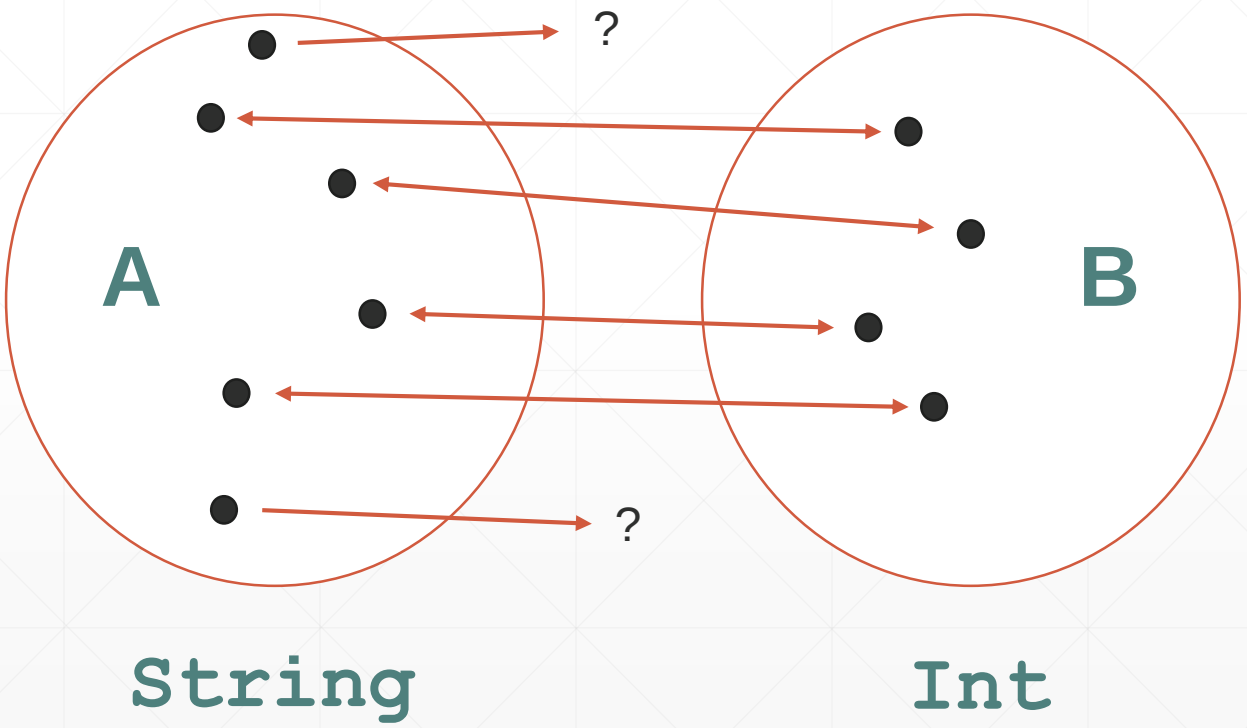
What if we don’t have such a nice correspondence?




```
String <-- Int
String --> Option[Int]
```

Prisms

What if we don't have such a nice correspondence?



`String` ⟵ `Int`

`String` ⟶ `Option[Int]`

Laws:

1. If there is an answer, going back must give the source.
2. If we go back, there must be an answer, which is the source.

Property Testing

Laws => Automated property testing

```
String <--- Int =

    _.toString

String ---> Option[Int] =

    Try(_.toInt).toOption
```

Property Testing

“ 

” .toInt = 9

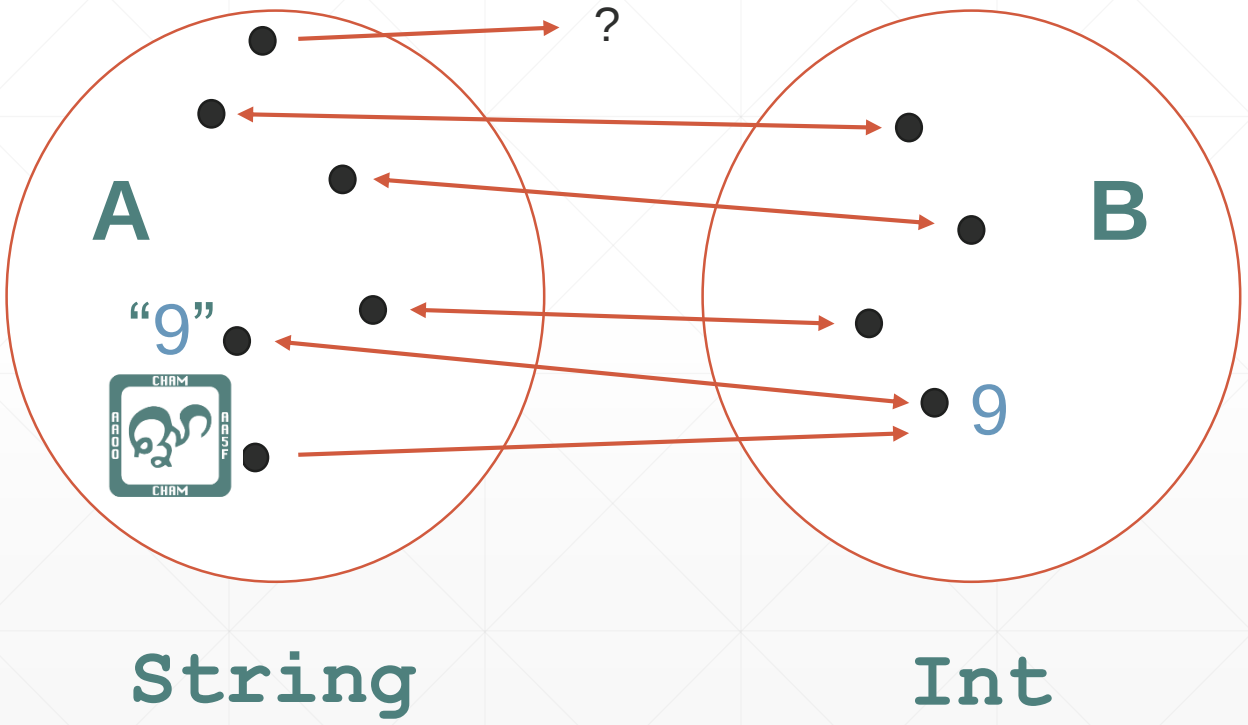
Property Testing

“ 

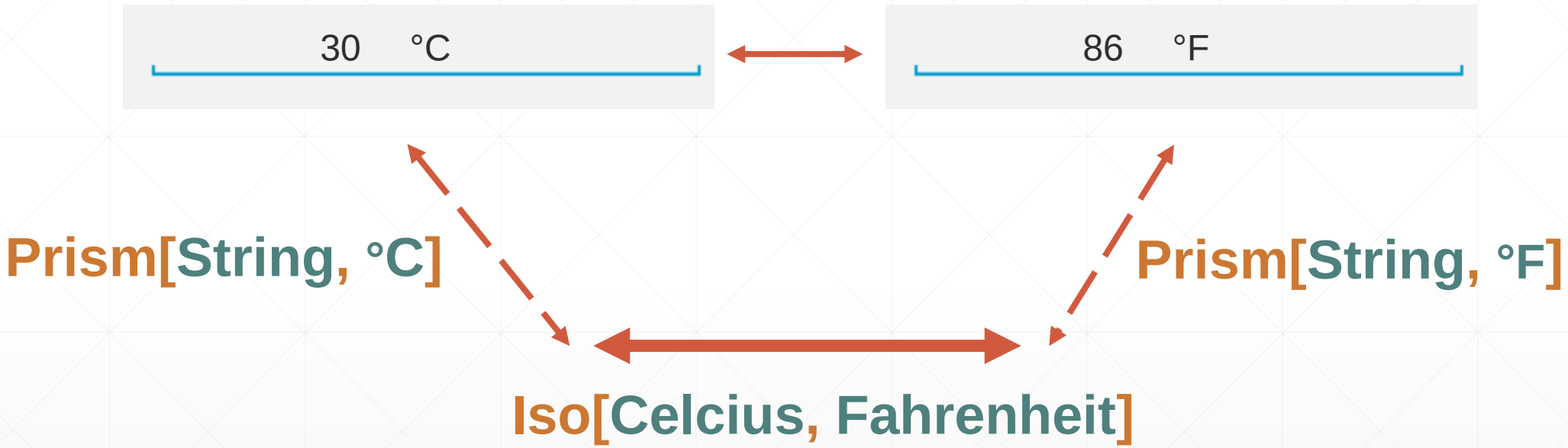
” .toInt = 9

WAT

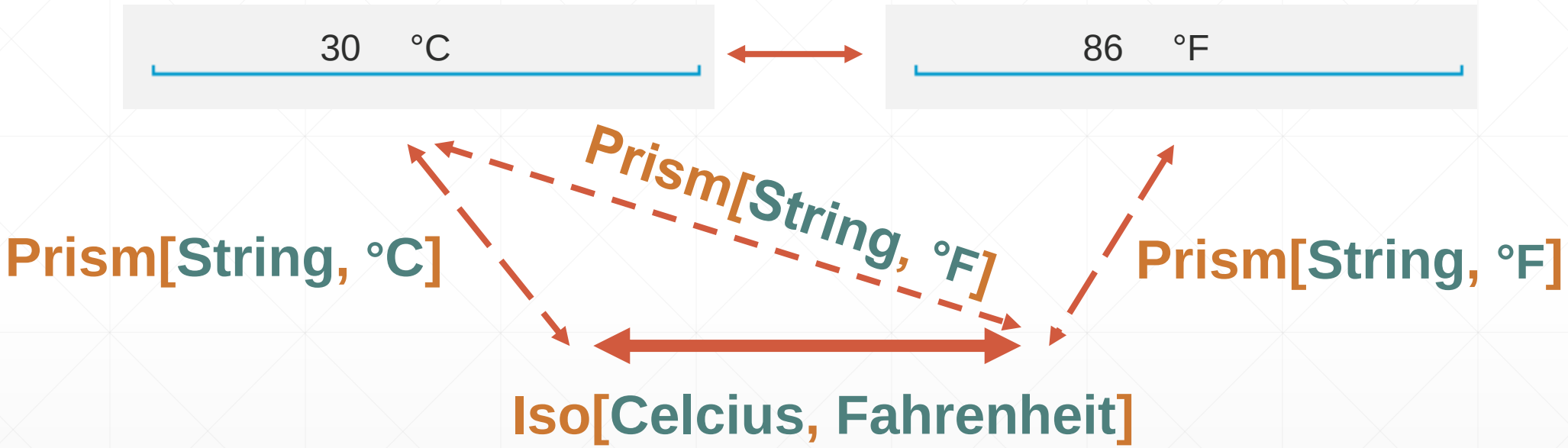
Prism Laws



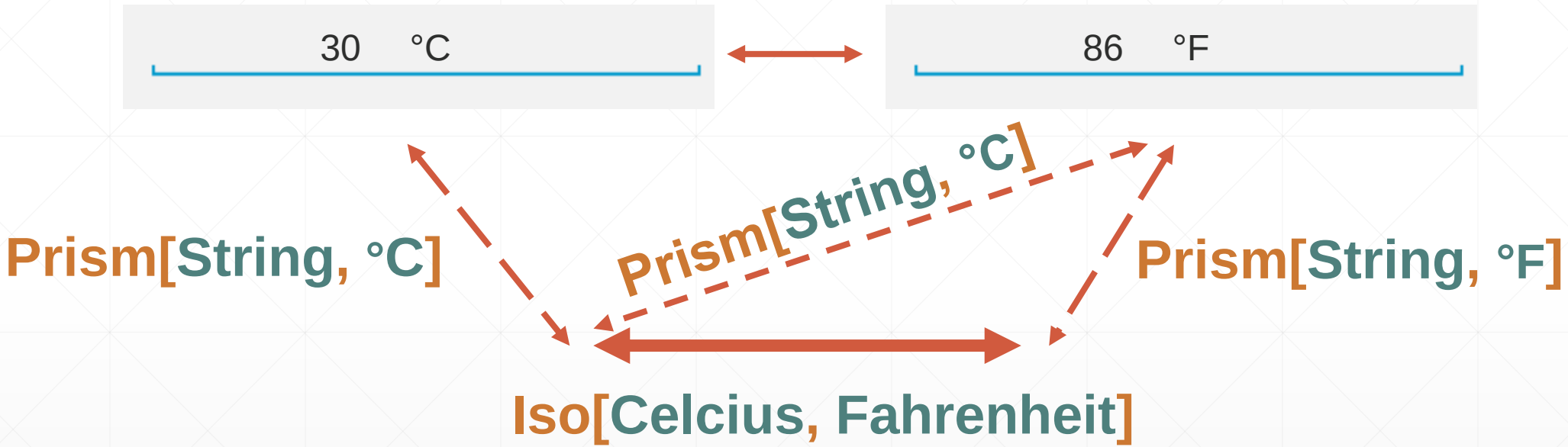
Example: Double binding



Example: Double binding



Example: Double binding



A bit of theory:
Van Laarhoven Lenses

Is it possible to unify all the lens functions?

- get: **S => A**
- set: **A => S => S**
- modify: **(A => A) => (S => S)**

A bit of theory:
Van Laarhoven Lenses

Is it possible to unify all the lens functions?

- get: **S => A**
- set: **A => S => S**
- modify: **(A => A) => (S => S)**
- modifyMaybe: **(A => Option[A]) => (S => Option[S])**
- modifyList: **(A => List[A]) => (S => List[S])**

Functors

```
trait Functor[F[_]]{
  def map[A,B](f: A => B)(fa: F[A]): F[B]
}
```

List Functor

```
trait Functor[F[_]]{
  def map[A,B](f: A => B)(fa: F[A]): F[B]
}
```

```
Functor[List]{
  def map[A,B](f: A => B)(list: List[A]): List[B] =
    list.map(f)
}
```

Option Functor

```
trait Functor[F[_]]{
  def map[A,B](f: A => B)(fa: F[A]): F[B]
}
```

```
Functor[Option]{
  def map[A,B](f: A => B)(opt: Option[A]): Option[B] = opt match {
    case None    => None
    case Some(a) => Some(f(a))
  }
}
```

Van Laarhoven Lenses

The answer:

```
Functor[F] => (A => F[A]) => (S => F[S])
```

Van Laarhoven Lenses

The answer:

```
Functor[F] => (A => F[A]) => (S => F[S])
```

- get: `S => A`
- set: `A => S => S`
- modify: `(A => A) => (S => S)`
- modifyMaybe: `(A => Option[A]) => (S => Option[S])`
- modifyList: `(A => List[A]) => (S => List[S])`

Van Laarhoven Lenses

The answer:

```
Functor[F] => (A => F[A]) => (S => F[S])
```

- get: `S => A`
- set: `A => S => S`
- modify: `(A => A) => (S => S)`
- modifyMaybe: `(A => Option[A]) => (S => Option[S])`
- modifyList: `(A => List[A]) => (S => List[S])`

Van Laarhoven Lenses

The answer:

```
lens: Functor[F] => (A => F[A]) => (S => F[S])
```

```
type Id[A] = A

lens: (A => Id[A]) => (S => Id[S])

lens: (A => A) => (S => S)

modify = lens[Id]
```

Identity Functor

```
trait Functor[F[_]]{
  def map[A,B](f: A => B)(fa: F[A]): F[B]
}
```

```
type Id[A] = A
Functor[Id]{
  def map[A,B] (f: A => B)(a: Id[A]): Id[B]
}
```

Identity Functor

```
trait Functor[F[_]]{
  def map[A,B](f: A => B)(fa: F[A]): F[B]
}
```

```
type Id[A] = A
Functor[Id]{
  def map[A,B] (f: A => B)(a: Id[A]): Id[B]
  def map[A,B] (f: A => B)(a: A): B = f(a)
}
```

Van Laarhoven Lenses

The answer:

```
lens: Functor[F] => (A => F[A]) => (S => F[S])
```

```
set(b) = modify(_ => b)
```

```
get: S => A = ???
```

Van Laarhoven Lenses

The answer:

lens: **Functor[F] => (A => F[A]) => (S => F[S])**

set(b) = modify(_ => b)

get: **S => A = ???**

Van Laarhoven Lenses

The answer:

```
lens: Functor[F] => (A => F[A]) => (S => F[S])
```

```
type Const[X][T] = X

F = Const[A]

lens: (A => Const[A][A]) => (S => Const[A][S])

lens: (A => A) => (S => A)

get = lens[Const[A]](a => a)
```

Const Functor

```
trait Functor[F[_]]{
  def map[A,B](f: A => B)(fa: F[A]): F[B]
}
```

```
type Const[X][T] = X
Functor[Const[X]]{
  def map[A,B] (f: A => B)(fa: Const[X][A]): Const[X][B]
}
```

Const Functor

```
trait Functor[F[_]]{
  def map[A,B](f: A => B)(fa: F[A]): F[B]
}
```

```
type Const[X][T] = X
Functor[Const[X]]{
  def map[A,B] (f: A => B)(fa: Const[X][A]): Const[X][B]
  def map[A,B] (f: A => B)(x: X): X = x
}
```

Van Laarhoven Lenses

The answer:

```
Functor[F] => (A => F[A]) => (S => F[S])
```

- get: `S => A`
- set: `A => S => S`
- modify: `(A => A) => (S => S)`
- modifyMaybe: `(A => Option[A]) => (S => Option[S])`
- modifyList: `(A => List[A]) => (S => List[S])`

Creating Van Laarhoven Lenses

lens: Functor[F] => (A => F[A]) => (S => F[S])

```
def get: S => A
def set: A => S => S
def lens[F[_]:Functor](f: A => F[A])(s: S): F[S] =
      f(get(s)).     map     (a => set(a)(s))
```

A — F[A]

(A => S) => F[A] => F[S]

(A => S)

F[S]

```
Shortly:
lens f s = f(get(s)).map(a => set(a)(s))
```

Creating Van Laarhoven Lenses

lens: Functor[F] => (A => F[A]) => (S => F[S])

```
type Id[A] = A
F = Id

lens: (A => A) => (S => S)
lens f s = f(get(s)).map(a => set(a)(s))
           [   A   ]     [    A => S   ]

modify f s = set(f(get(s)))(s)
set x s = modify(_ => x) s
        = set(x)(s)
```

Creating Van Laarhoven Lenses

lens: Functor[F] => (A => F[A]) => (S => F[S])

```
type Const[A][X] = A
F = Const[A]

lens: (A => A) => (S => A)
lens f s = f(get(s)).map(a => set(a)(s))
           [   A   ]     [    A => S   ]

get' f s = f(get(s))
get s = get'(id)(s) = get(s)
```

Monocle

- Provides lots of built-in optics and functions
- Macros for creating lenses
- Monocle used different models of lenses over time
- Current lens representation: PLens – Van Laarhoven hybrid.
- Performance: limited by scala function values – pretty good.

Resources

- Monocle on github
- Simon Peyton Jones's lens talk at Scala Exchange 2013
- Edward Kmett on Lenses with the State Monad

Thank you!

Extra Slides

ASTs - Lenses for APIs

We can simplify our mental model with lenses

```
case class Person(fullName: String, address: Address)
```

Case class decleration

Class name

List[Field name -> Type]

ASTs - Lenses for APIs

We can simplify our mental model with lenses

Public constructor

val

```
case class Person(fullName: String, address: Address)
```

Case class decleration

Class name

List[Field name -> Type]

Lens[Complex model, Simple model]

Polymorphic Optic Instances

How can we use the same lens for many types?

```
first.set("Hello")((1,2))       == ("Hello",2)
first.set("Hello")((1,2,3))     == ("Hello",2,3)
first.set("Hello")((1,2,3,4)) == ("Hello",2,3,4)
```

Example: Json

The structure of a specific Json object isn't defined at the type system;
Each element can be a **String** or a **Number** or an **Array** or an **Object**.
We want to process Json assuming our specific structure, and let the system handle failures for us.
We can define **Prism[Json, String]**, **Prism[Json, Double]**, **Prism[Json, List[Json]]** and **Prism[Json, Map[String, Json]]**.
Now we can compose these prisms to go deep inside a Json object and manipulate it.

* Polymorphic Lens Composition