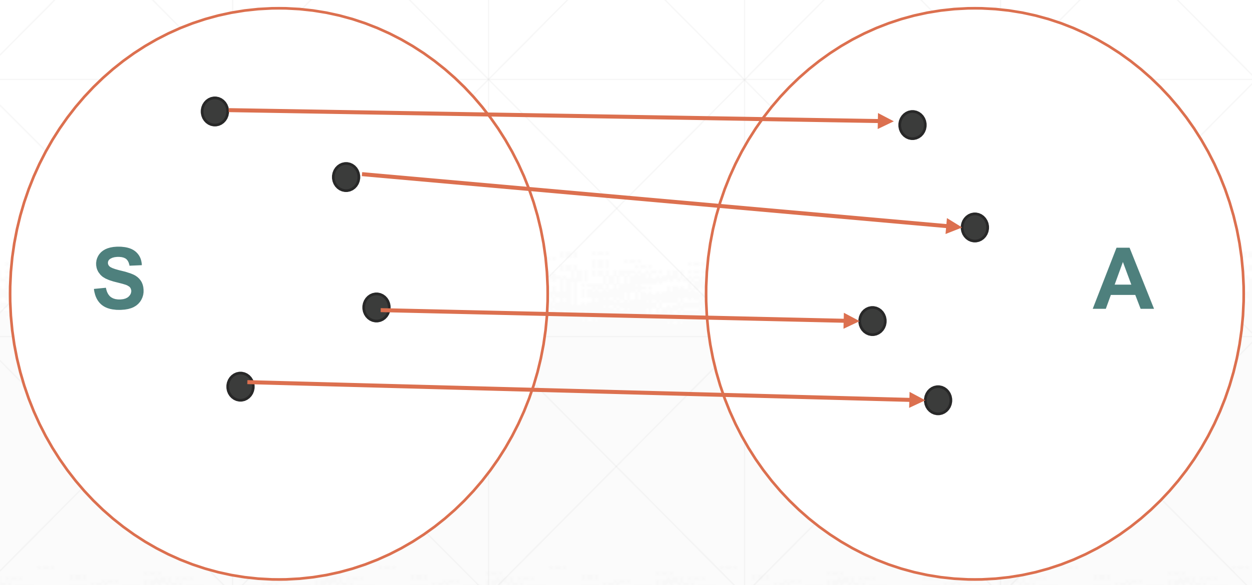
Beyond Scala Lenses

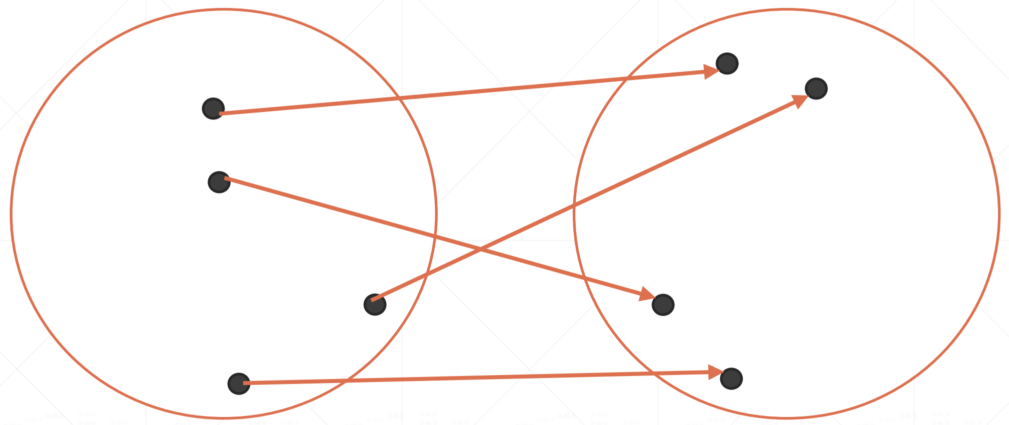
By Julien Truffaut (@julien-truffaut on github – twitter - IRC)

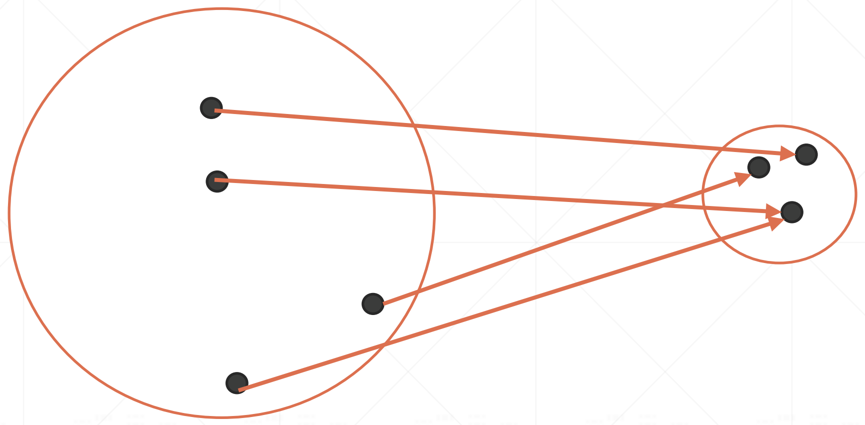
Function

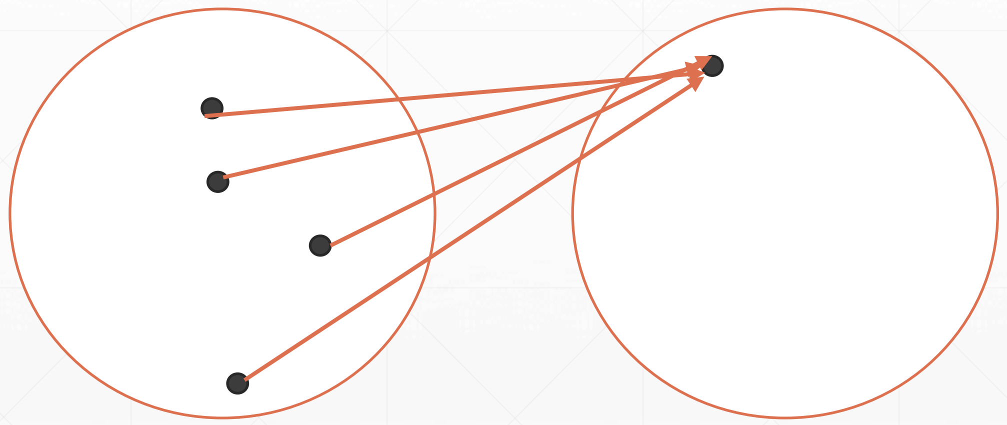
A function transform all s in S into an A



Function



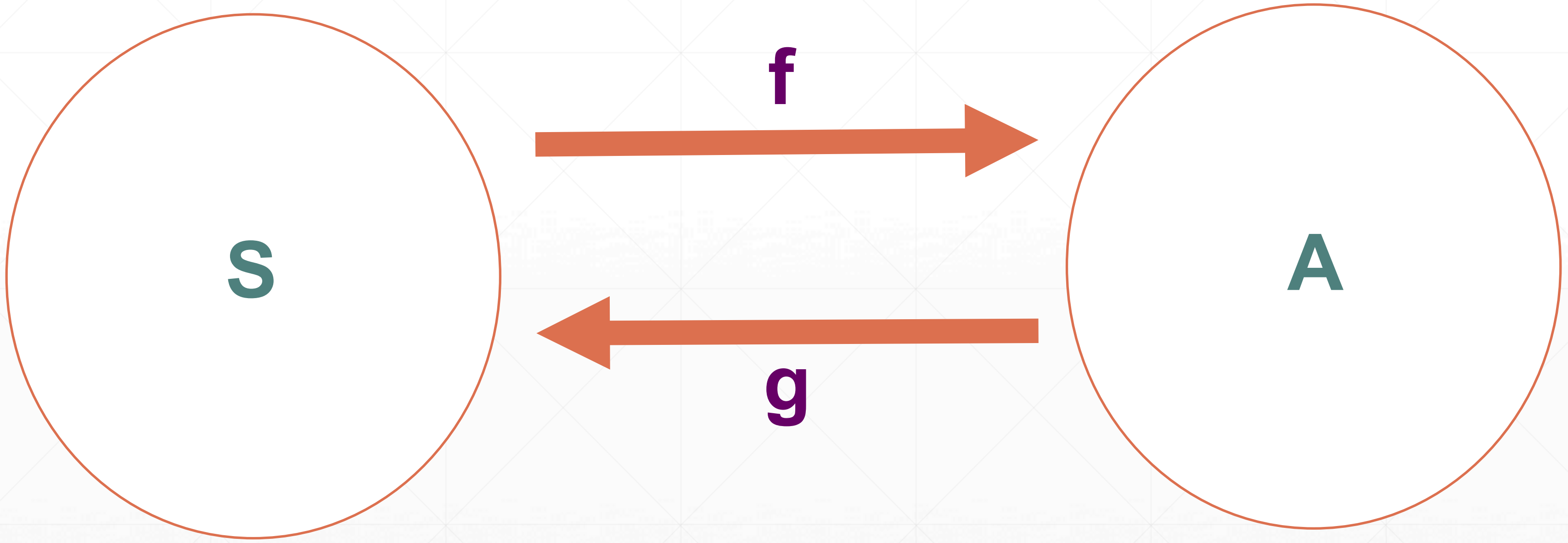




Pair of Functions

g(f(s)) = ???: S

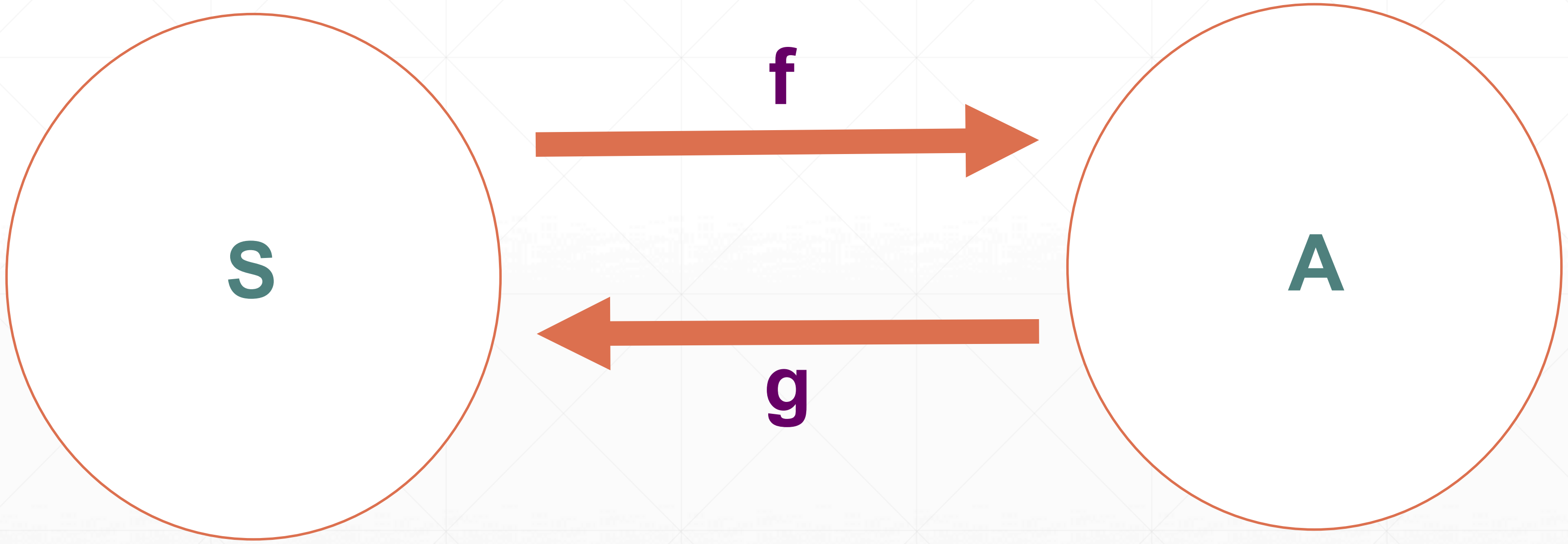
f(g(a)) = ???: A



Isomorphism

For all s: S, g(f(s)) == s

For all a: A, f(g(a)) == a



Iso

```
case class Iso[S,A](
  get       : S => A,
  reverseGet: A => S
)
```

Properties:
For all s: S, reverseGet(get(s)) == s
For all a: A, get(reverseGet(a)) == a

Modify

$$\begin{tikzcd} A \arrow[r, "m"] & A \end{tikzcd}$$

Modify

$$\begin{tikzcd} A \arrow[r, "m"] & A \arrow[d, "reverseGet"] \\ S \arrow[u, "get"] & S \end{tikzcd}$$

Modify

$$\begin{tikzcd}
A \arrow[r, "m"] & A \arrow[d, "\text{reverseGet}"] \\
S \arrow[u, "\text{get}"] \arrow[r, dashed, "\text{modify}"'] & S
\end{tikzcd}$$

Compose

$$\begin{tikzcd}
S \arrow[r, leftrightarrow, "\text{Iso}"] \arrow[rr, leftrightarrow, dashed, bend right, "\text{Iso}"'] & A \arrow[r, leftrightarrow, "\text{Iso}"] & B
\end{tikzcd}$$

Iso Derived Methods

```scala
case class Iso[S,A](
  get       : S => A,
  reverseGet: A => S
){
  def modify(m: A => A)(s: S): S
  def compose[B](other: Iso[A,B]): Iso[S,B]
  def reverse: Iso[A,S]
}
```

Distance

```
class Robot{
  def moveBy(d: Double): Robot
}

val nono: Robot = …

nono.moveBy(100.5) // Meters

nono.moveBy(3)      // Kilometers

nono.moveBy(-2.5)   // Yards
```

Distance Safe

```
case class Meter(d: Double)
case class Yard(d: Double)

class Robot{
  def moveBy(m: Meter): Robot
}

nono.moveBy(Meter(100.5))
nono.moveBy(Yard(3.0))    // does not compile
```

Iso

Meter $\longleftrightarrow$ Yard

Iso

$$\begin{tikzcd}
\text{Meter} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \text{Yard} \\
\text{Kilometer} \arrow[ur, leftrightarrow, dashed] &
\end{tikzcd}$$

Iso

$$\begin{tikzcd}
\text{Meter} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow, dashed] & \text{Yard} \arrow[d, leftrightarrow] \arrow[dl, leftrightarrow, dashed] \\
\text{Kilometer} \arrow[r, leftrightarrow, dashed] & \text{Mile}
\end{tikzcd}$$

Iso Example

```
val meterToYard: Iso[Meter, Yard] = Iso(
  m =>  Yard(m.value * 1.09361),
  y => Meter(y.value * 0.9144),
)

meterToYard.get(Meter(200)) == Yard(218.7219999…)

nono.moveBy(meterToYard.reverseGet(Yard(2.5))
```

Iso Composition

```
case class KiloMeter(value: Double)
case class Mile(value: Double)

val meterToKilometer: Iso[Meter, KiloMeter] = …
val yardToMile       : Iso[Yard, Mile] = …

val kilometerToMile: Iso[KiloMeter, Mile] =
  meterToKilometer.reverse compose
  meterToYard compose
  yardToMile
```

Other Iso

```
def listToVector[A]: Iso[List[A], Vector[A]]

val stringToList: Iso[String, List[Char]]


case class Person(name: String, age: Int)

val personToTuple: Iso[Person, (String, Int)]
```

Iso Properties

For all s: S, reverseGet(get(s)) == s

For all a: A, get(reverseGet(a)) == a

Scalacheck

```
def isoLaws[S,A](iso: Iso[S,A]) = new Properties {
  property("One Way") = forAll { s: S =>
   iso.reverseGet(iso.get(s)) == s
  }
  property("Other Way") = forAll { a: A =>
   iso.get(iso.reverseGet(a)) == a
  }
}
```

Scalacheck

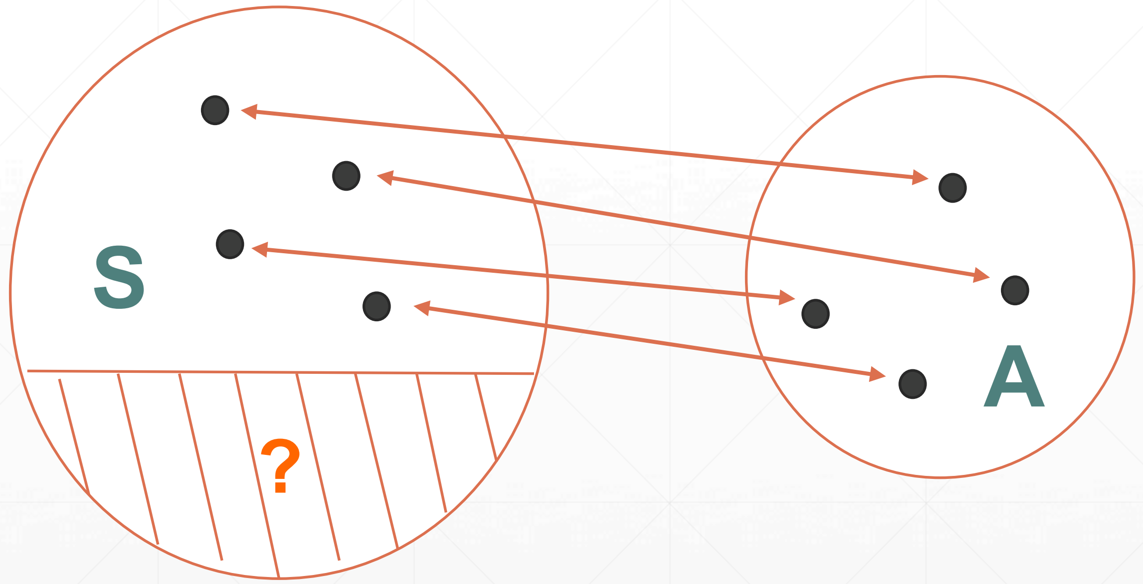
```scala
import org.spec2.scalaz.Spec

class IsoSpec extends Spec {
  checkAll(isoLaws(meterToYard))
}
```

Scalacheck

```
A counter-example is:
   [Meter(-1.0)] (after 0 try)



scala> meterToYard.reverseGet(meterToYard.get(Meter(-1)))
scala> res0: Meter = Meter(-0.9999969839999999)
```

Relax Isomorphism

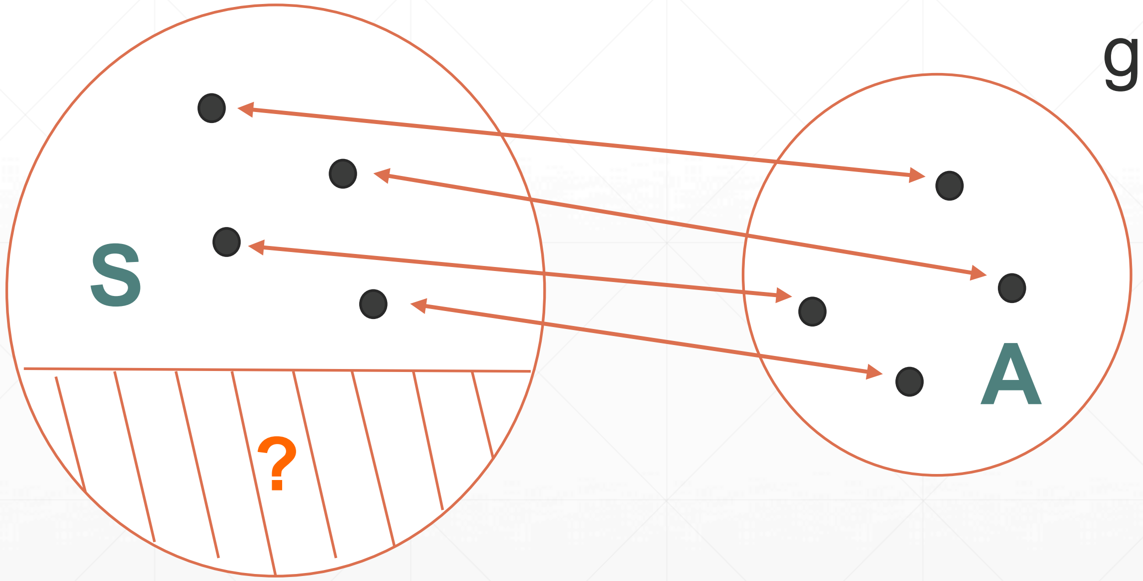
For all s: S such as f(s) exists, g(f(s)) == s
For all a: A, f(g(a)) == a



Relax Isomorphism

For all s: S such as f(s) exists, g(f(s)) == s
For all a: A, f(g(a)) == a

f is a Function[S, Option[A]]
g is a Function[A, S]



Prism

```
case class Prism[S,A](
  getOption : S => Option[A],
  reverseGet: A => S
)
```

Properties:
For all s: S, getOption(s) map reverseGet == Some(s) || None
For all a: A, getOption(reverseGet(a)) == Some(a)

Pattern matching

```
sealed trait List[A]
case class Cons[A](h: A, t: List[A]) extends List[A]
case class Nil[A]() extends List[A]

Cons.unapply(List(1,2,3)) == Some((1, List(2,3)))
Cons.unapply(Nil)         == None

Cons.apply(1, List(2,3))  == List(1,2,3)
```

Prism

```scala
sealed trait List[A]
case class Cons[A](h: A, t: List[A]) extends List[A]
case class Nil[A]() extends List[A]

def cons[A]: Prism[List[A], (A, List[A])]

cons.getOption(List(1,2,3)) == Some((1, List(2,3)))
cons.getOption(Nil)         == None

cons.reverseGet(1, List(2,3)) == List(1,2,3)
```

Prism Derived Methods

```
case class Prism[S,A](
  getOption : S => Option[A],
  reverseGet: A => S
){
  def modifyOption(f: A => A): S => Option[S]
  def modify(f: A => A): S => S
  def compose[B](other: Prism[A,B]): Prism[S,B]
  def compose[B](other: Iso[A,B]  ): ???[S,B]
}
```

Iso – Prism

Optic	f	g
Iso	S => A	A => S
Prism	S => Option[A]	A => S

```
def isoToPrism[S,A](iso: Iso[S,A]): Prism[S,A] =
  Prism(
    getOption   = s => Some(iso.get(s)),
    reverseGet = iso.reverseGet
  )
```

Iso – Prism

```
case class Prism[S,A]{
  def compose[B](other: Prism[A,B]): Prism[S,B]
  def compose[B](other: Iso[A,B]  ): Prism[S,B]
}

case class Iso[S,A]{
  def compose[B](other: Iso[A,B]  ):   Iso[S,B]
  def compose[B](other: Prism[A,B]): Prism[S,B]
}
```

Json

```scala
sealed trait Json
case class JNumber(v: Double) extends Json
case class JString(s: String) extends Json

val jNum: Prism[Json, Double] = …

jNum.modify(_ + 1)(JNumber(2.0)) == JNumber(3.0)
jNum.modify(_ + 1)(JString(“”))  == JString(“”)

jNum.modifyOption(_ + 1)(JString(“”)) == None
```

Safe Down Casting

```
def doubleToInt: Prism[Double, Int] = …

doubleToInt.getOption(3.4) == None
doubleToInt.getOption(3.0) == Some(3)

doubleToInt.reverseGet(5) == 5.0
```

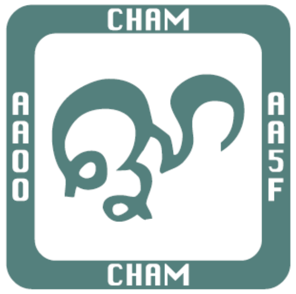
Prism Composition

```
sealed trait Json
case class JNumber(v: Double) extends Json
case class JString(s: String) extends Json

val jInt = jNum compose doubleToInt

jInt.getOption(Jnumber(3.0)) == Some(3)
jInt.getOption(Jnumber(5.9)) == None

jInt.getOption(JString(“”))  == None
```

Where is the bug?

```
def stringToInt: Prism[String, Int] = Prism(
  getOption  = s => Try(s.toInt).toOption,
  reverseGet = _.toString
)
stringToInt.modify(_ * 2)("5") == "10"

stringToInt.getOption("5")   == Some(5)
stringToInt.getOption("-3")  == Some(-3)
stringToInt.getOption("5.7") == None
stringToInt.getOption("99999999999999999") == None
stringToInt.getOption("Hello") == None
```

Tadam !

“ 

 ”.toInt = 9

Lens

```
case class Lens[S,A](
  get:     S   => A,
  set:(A,  S)  => S
)
```

Properties:
For all s: S, set(get(s), s) == s
For all a: A, s: S, get(set(a, s)) == a

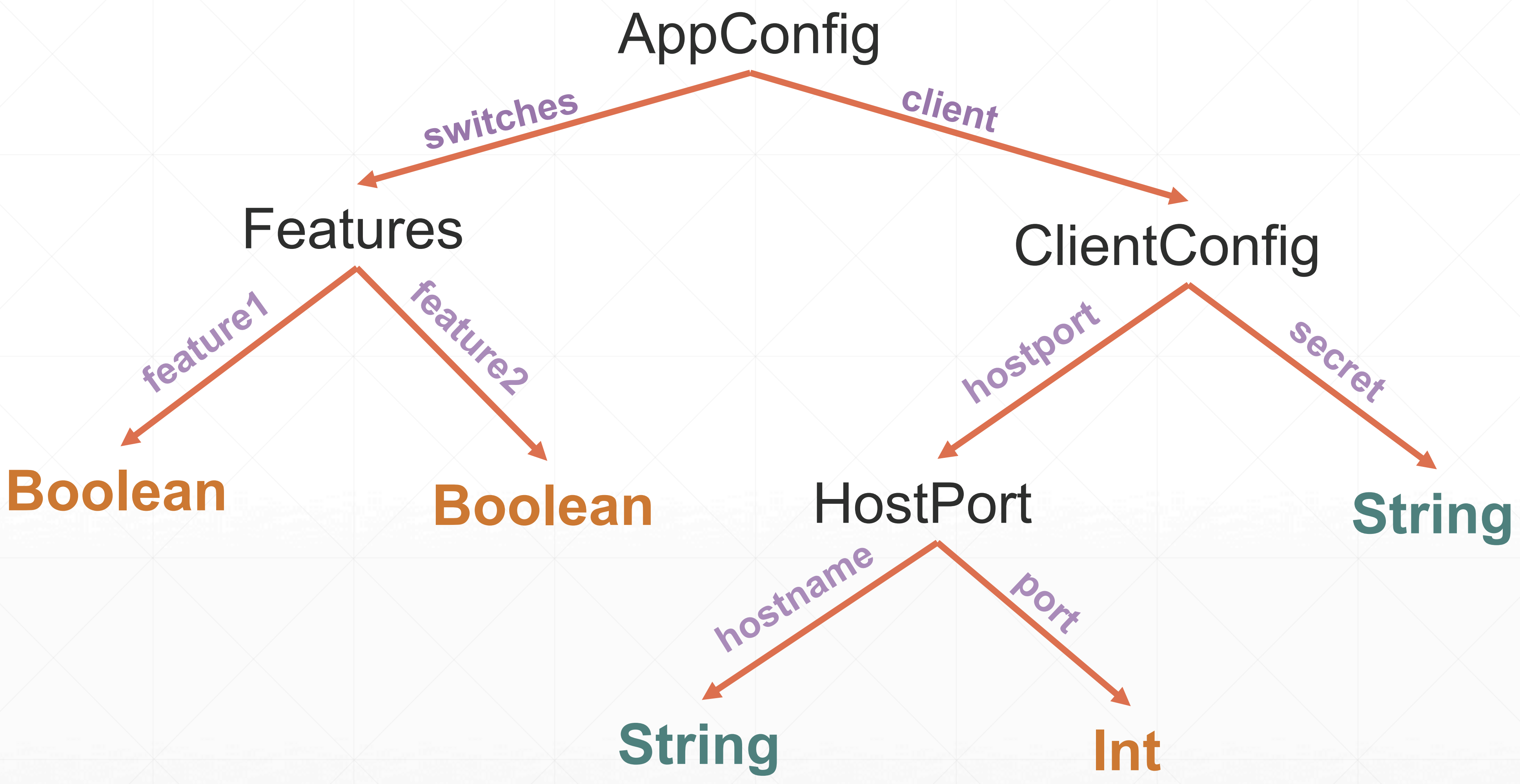
Lens Derived Methods

```
case class Lens[S,A](
  get:     S  => A,
  set:(A, S)  => S
){
  def modify(f: A => A): S => S
  def compose[B](other: Lens[A,B]): Lens[S,B]
  def compose[B](other: Iso[A,B] ): Lens[S,B]
}
```

Iso – Lens

Optic	f	g
Iso	S => A	A => S
Lens	S => A	(A, S) => S

```
def isoToLens[S,A](iso: Iso[S,A]): Lens[S,A] =
  Lens(
    get = iso.get,
    set = (a, _) => iso.reverseGet(a)
  )
```


AppConfig
switches
client
Features
ClientConfig
feature1
feature2
hostport
secret
Boolean
Boolean
HostPort
String
hostname
port
String
Int

Nested Objects

```
val config = AppConfig(…)

config.client.hostport.port // 9999

config.copy(
  client = config.client.copy(
    hostport = config.client.hostport.copy(
      port = 8000
)))
```

Nested Objects with Lenses

```scala
val client: Lens[AppConfig, ClientConfig] = …
val hostPort: Lens[ClientConfig, HostPort] = …
val port: Lens[HostPort, Int] = …

(client compose hostPort compose port)
  .get(config) // 9999

(client compose hostPort compose port)
  .set(8000, config)
```

Boiler plate

```
def toogleFeature1(config: AppConfig): AppConfig =
  config.copy(
    features = config.features.copy(
      feature1 = ! config.features.feature1
  ))

def toogleFeature2(config: AppConfig): AppConfig = …
```

Lenses as a Pointer

```
def toogle(bool: Lens[Feature, Boolean])
   : AppConfig => AppConfig =
  (features compose bool).modify(!_)

toogle(feature1)(config)
toogle(feature2)(config)

def toogleAll: AppConfig => AppConfig =
   toogle(feature1) . toogle(feature2)
```

At

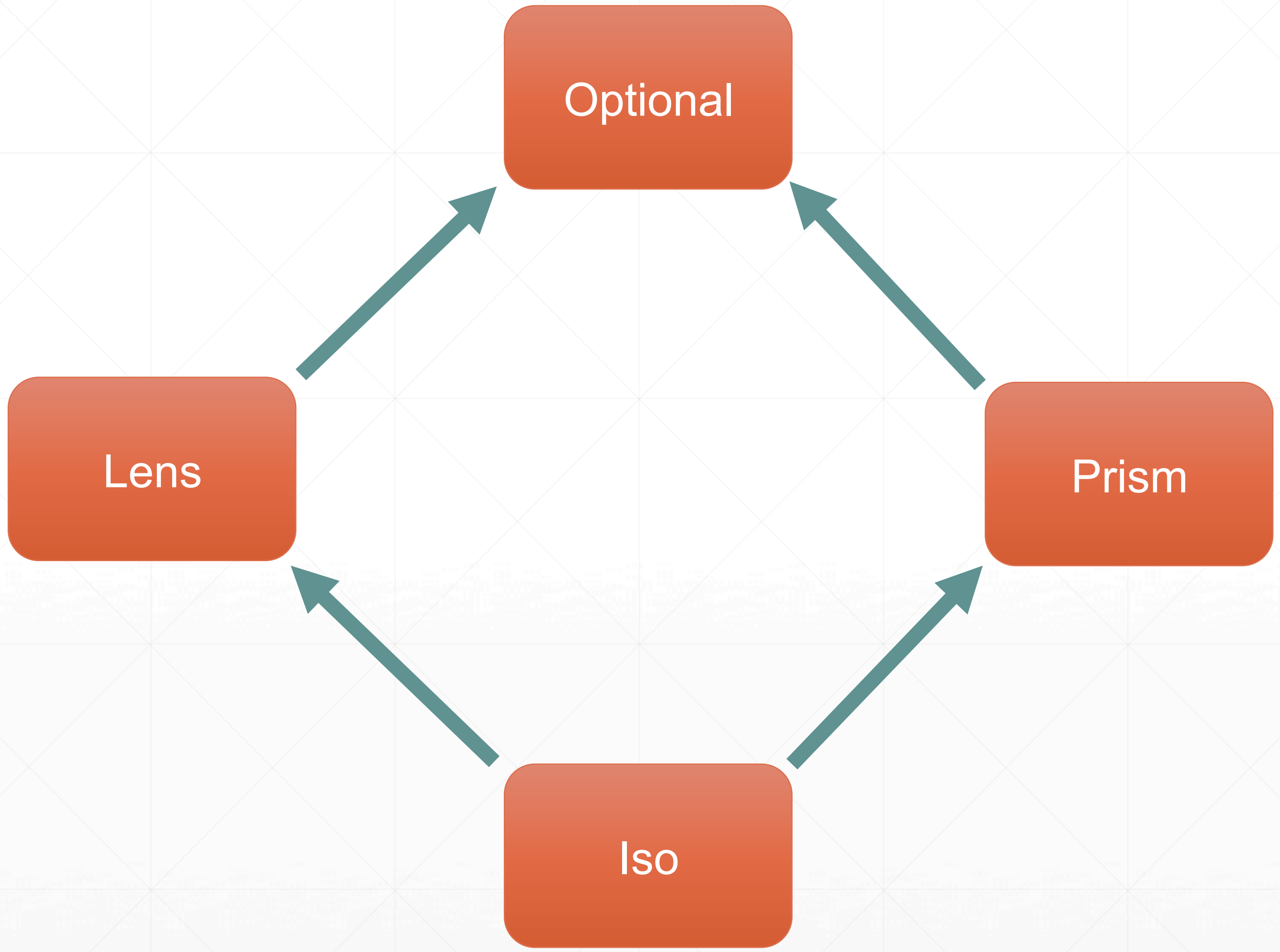
```scala
def at(k: K)[K,V]: Lens[Map[K,V], Option[V]] = Lens(
  get = _.get(k),
  set = (v, m) => m + (k -> v)
)
val m = Map(1 -> "one", 2 -> "two")
at(2).get(m)== Some("two")
at(9).get(m)== None

at(1).set(None, m)== Map(2 -> "two")
at(3).set(Some("three"), m)==
 Map(1 -> "one", 2 -> "two" , 3 -> "three")
```

What Next?

Optic	f	g
Iso	S => A	A => S
Prism	S => Option[A]	A => S
Lens	S => A	(A, S) => S

Optional

Optic	f	g
Iso	S => A	A => S
Prism	**S => Option[A]**	A => S
Lens	S => A	**(A, S) => S**
Optional	S => Option[A]	(A, S) => S


Optional
Lens
Prism
Iso

Optional

```
case class Optional[S,A](
  getOption: S => Option[A],
  set      :(A, S) => S
)
```

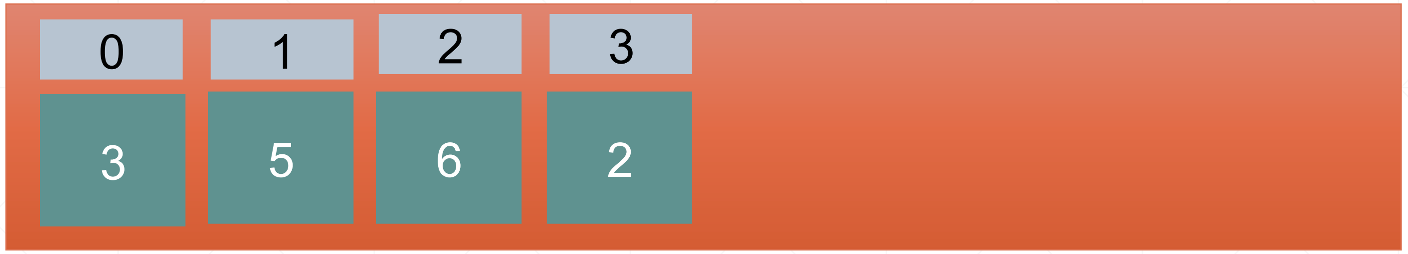
Properties:
For all s: S, getOption(s) map set(_, s) == s
For all a: A, s: S, getOption(set(a, s)) == Some(a) || None

Index

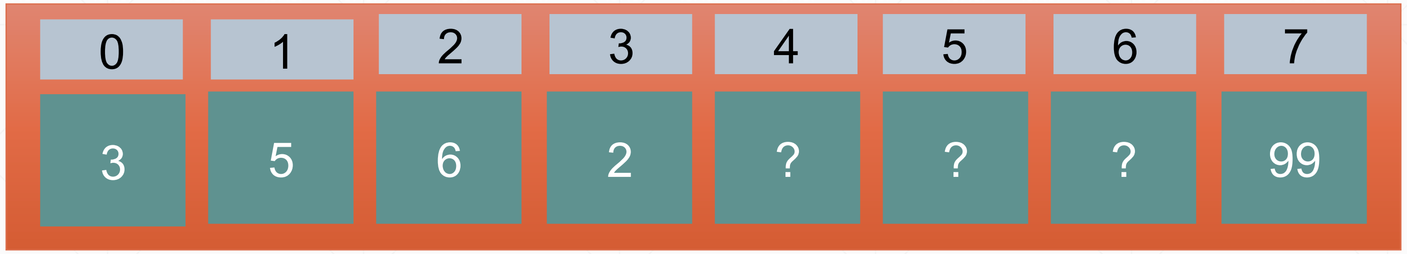
```
def indexL[A](i: Int): Optional[List[A], A] = …

indexL(1).getOption(List(1,2,3)) == Some(2)
indexL(5).getOption(List(1,2,3)) == None

indexL(2).modify(_ + 1)(List(1,2,3)) == List(1,3,3)

def indexV(i: Int): Optional[Vector[A], A] =
  vectorToList compose indexL(i)
```

Index != At

```
val l = List(3,5,6,2)
```



```
index(0).set(99, l)
```



Complete API

```
List(1,2,3).headOption == Some(1)

def headOption[A]: Optional[List[A], A] = …

headOption.getOption(List(1,2,3)) == Some(1)
headOption.set(0,List(1,2,3)) == List(0,2,3)

headOption.setOption(0,List(1)) == Some(List(0))
headOption.setOption(0,Nil)     == None
```

Study Case: Json

```scala
sealed trait Json

case class JNumber(v: Double) extends Json
case class JString(s: String) extends Json
case class JArray(l: List[Json]) extends Json
case class JObject(m: Map[String,Json]) extends Json
```

Study Case: Json

```
val john = “““
{
  “first_name” : “John”,
  “last_name”  : “Doe”,
  “age”        : 26,
  “siblings”   : [
    { “first_name” : “Zoe” },
    { “first_name” : “Bill”, “age” : 23 },
  ]
}
”””
```

Set Zoe’s Age

$$\begin{tikzcd}
\text{String} \arrow[r, "\text{parse}"] & \text{Json} \arrow[r, "\text{jObject}"] & \text{Map[String, Json]} \arrow[d, "\text{index(“siblings”)}"] \\
\text{Option[Json]} & & \text{Json} \arrow[d, "\text{jArray}"] \\
\text{Map[String, Json]} \arrow[u, "\text{at(“age”)}"] & \text{Json} \arrow[l, "\text{jObject}"'] & \text{List[Json]} \arrow[l, "\text{headOption}"']
\end{tikzcd}$$

Parse

```
def stringToJson: ???[String, Json] = …

def parse(s: String): Option[Json] = …
def toString(json: Json): String= …
```

Parse

```
def parse: ???[String, Json] = …

def fromString(s: String): Option[Json] = …
def toString(json: Json): String = …
```

Parse

```
def parse: Prism[String, Json] = …

parse.getOption("""{ a : 3 }""")
parse.getOption ("""{a:3}""")

For all s: String,
  getOption(reverseGet(s)) == Some(s) || None

==> def parse: Optional[String, Json] = …
```

Set Zoe’s Age

$$\begin{tikzcd}
\text{String} \arrow[r, "\text{parse}", "\text{Optional}"'] & \text{Json} \arrow[r, "\text{jObject}"] & \text{Map[String, Json]} \arrow[d, "\text{index(“siblings”)}"] \\
\text{Option[Json]} & & \text{Json} \arrow[d, "\text{jArray}"] \\
\text{Map[String, Json]} \arrow[u, "\text{at(“age”)}"] & \text{Json} \arrow[l, "\text{jObject}"] & \text{List[Json]} \arrow[l, "\text{headOption}"]
\end{tikzcd}$$

Sum Type

```
def jNumber: Prism[Json, Int] = …
def jString: Prism[Json, String] = …
def jArray : Prism[Json, List[Json]] = …
def jObject: Prism[Json, Map[String, Json]] = …
```

Set Zoe’s Age

$$\begin{tikzcd}
\text{String} \arrow[r, "\text{parse}", "\text{Optional}"'] & \text{Json} \arrow[rr, "\text{jObject}", "\text{Prism}"'] & & \text{Map[String, Json]} \arrow[d, "\text{index(“siblings”)}"] \\
\text{Option[Json]} & & & \text{Json} \arrow[d, "\text{jArray}", "\text{Prism}"'] \\
\text{Map[String, Json]} \arrow[u, "\text{at(“age”)}"] & & \text{Json} \arrow[ll, "\text{Prism}"', "\text{jObject}"] & \text{List[Json]} \arrow[l, "\text{headOption}"]
\end{tikzcd}$$

Set Zoe’s Age

$$\begin{tikzcd}
\text{String} \arrow[r, "\text{parse}", "\text{Optional}"'] & \text{Json} \arrow[r, "\text{jObject}", "\text{Prism}"'] & \text{Map[String, Json]} \arrow[d, "\text{index(“siblings”)}", "\text{Optional}"'] \\
\text{Option[Json]} & & \text{Json} \arrow[d, "\text{jArray}", "\text{Prism}"'] \\
\text{Map[String, Json]} \arrow[u, "\text{Lens}"', "\text{at(“age”)}"] & \text{Json} \arrow[l, "\text{Prism}"', "\text{jObject}"] & \text{List[Json]} \arrow[l, "\text{headOption}"]
\end{tikzcd}$$

Set Zoe’s Age

```
(parse
  compose jObject
  compose index(“siblings”)
  compose jArray
  compose headOption
  compose jObject
  compose at(“age”)
).set(JNumber(28))(john)
```

Study Case: Json

```
"""
{
  "first_name" : "John",
  "last_name"  : "Doe",
  "age"        : 26,
  "siblings"   : [
    { "first_name" : "Zoe" , "age" : 28 },
    { "first_name" : "Bill", "age" : 23 },
  ]
}
"""
```

Monocle

- Provides lots of built-in optics and functions
- Macros for creating Lenses, soon Iso and Prism
- Efficient implementation for Scala
- More boiler plate than in haskell due to weaker type system and type inference

Resources

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

Acknowledgement

- Member Monocle on gitter and irc for advice and review
- Special thanks to @NightRa for helps with slides and content

Thank you!