An Introduction to Optics at the *Funktionaler Stammtisch*, Leipzig

Sonat Süer

Soostone Inc.

April 26th, 2023

About me

I

- have a background in mathematics –PhD from UIUC in mathematical logic,
- worked in academia for a couple of years,
- have been writing Haskell for a living for the last $\sim$6 years.

About the talk

The talk will

- focus on simple optics, that is, optics with only two type parameters as opposed to four

About the talk

The talk will

- focus on simple optics, that is, optics with only two type parameters as opposed to four
- be mostly language independent but examples will be in Kotlin,

About the talk

The talk will

- focus on simple optics, that is, optics with only two type parameters as opposed to four
- be mostly language independent but examples will be in Kotlin,
- only cover isomorphisms, lenses and prisms,

About the talk

The talk will

- focus on simple optics, that is, optics with only two type parameters as opposed to four
- be mostly language independent but examples will be in Kotlin,
- only cover isomorphisms, lenses and prisms,
- be about ‘what’ rather than ‘how’, since the Arrow-Optics library already has enough documentation about usage.

Optics Generally

Suppose you need to manipulate complex immutable data structures in your code. A naive approach to this problem may easily get out of hand –excessive boilerplate, code repetition, incomposable design, etc. . . Optics offer a principled and compact solution to this problem.

Optics Generally

Suppose you need to manipulate complex immutable data structures in your code. A naive approach to this problem may easily get out of hand –excessive boilerplate, code repetition, incomposable design, etc. . . Optics offer a principled and compact solution to this problem.

In the context of Kotlin, this usually means avoiding boilerplate using reflection and writing compact code without using the `copy` function.

Isomorphisms

The notion of isomorphism is an abstraction which captures the idea of representation independence.

Isomorphisms

The notion of isomorphism is an abstraction which captures the idea of representation independence.

More precisely, an isomorphism between two types –or sets if you will– A and B is a pair of functions $f\colon A \to B$ and $g\colon B \to A$ such that

$$g(f(a)) = a \quad \text{and} \quad f(g(b)) = b$$

for all a in A and b in B.

Isomorphisms

The notion of isomorphism is an abstraction which captures the idea of representation independence.

More precisely, an isomorphism between two types –or sets if you will– A and B is a pair of functions $f \colon A \to B$ and $g \colon B \to A$ such that

$$g(f(a)) = a \quad \text{and} \quad f(g(b)) = b$$

for all a in A and b in B.

This means that we can go back and forth between A and B without loosing any information.

Examples of Isomorphisms

Let us look at an example implemented in Kotlin using the Arrow-Optics library:

```
data class Celsius(val rawValue: Double)
data class Fahrenheit(val rawValue: Double)

val celsiusIsoFahrenheit: Iso<Celsius,Fahrenheit> =
  Iso(
    get = { celsius ->
      Fahrenheit(celsius.rawValue * 1.8 + 32.0) },
    reverseGet = { fahrenheit ->
      Celsius((fahrenheit.rawValue - 32.0) / 1.8) }
  )
```

Here f and g correspond to `get` and `reverseGet`, respectively.

Examples of Isomorphisms

Now using this isomorphism we can jump between Celsius and Fahrenheit. For instance:

```
// Defined with Celsius in mind
fun isFreezingCelsius(celsius: Celsius) : Boolean {
  return celsius.rawValue < 0
}
```

Examples of Isomorphisms

Now using this isomorphism we can jump between Celsius and Fahrenheit. For instance:

```
// Defined with Celsius in mind
fun isFreezingCelsius(celsius: Celsius) : Boolean {
  return celsius.rawValue < 0
}

// Used with Fahrenheit
val temperatureInFahrenheit = Fahrenheit(30.0);
val isTemperatureFreezing =
  isFreezingCelsius(
    celsiusIsoFahrenheit.reverseGet(
        temperatureInFahrenheit)
  )
```

Examples of Isomorphisms

This is nice but it goes in only one direction. Here is a general purpose combinator which uses both functions in an isomorphism:

```
fun <A,B> asIf(iso: Iso<A,B>, operation: (A) -> A):
  (B) -> B {
    return { b ->
      iso.get(operation(iso.reverseGet(b))) }
}
```

Examples of Isomorphisms

This is nice but it goes in only one direction. Here is a general purpose combinator which uses both functions in an isomorphism:

```
fun <A,B> asIf(iso: Iso<A,B>, operation: (A) -> A):
  (B) -> B {
    return { b ->
      iso.get(operation(iso.reverseGet(b))) }
}

// Defined with Celsius in mind
fun increaseCelsius(celsius: Celsius): Celsius {
  return Celsius(celsius.rawValue + 10)
}

// Used with Fahrenheit
val increased: Fahrenheit =
  asIf(celsiusIsoFahrenheit)
    {c -> increaseCelsius(c)}(temperatureInFahrenheit)
```

Inverse of an Isomorphism

Jolly good! We have operations defined in terms of Celsius and we can use them with Fahrenheit. But what if we have something designed for Fahrenheit and we need to use it with Celsius? Do we have to define a new isomorphism?

Inverse of an Isomorphism

Jolly good! We have operations defined in terms of Celsius and we can use them with Fahrenheit. But what if we have something designed for Fahrenheit and we need to use it with Celsius? Do we have to define a new isomorphism?

No! We have this general purpose function:

```
fun <A,B> inverseIso(iso : Iso<A,B>): Iso<B,A>{
  return Iso(
    get = { b -> iso.reverseGet(b) },
    reverseGet = { a -> iso.get(a) }
  )
}
```

Composition of Isomorphisms

What happens if we have more than two representations?

```
data class Kelvin(val rawValue: Double)

val celsiusIsoKelvin: Iso<Celsius,Kelvin> =
  Iso(
    get = { celsius ->
      Kelvin(celsius.rawValue + 273.0) },
    reverseGet = { kelvin ->
      Celsius(kelvin.rawValue - 273.0) }
  )
```

Do we also need to define an isomorphism between Fahrenheit and Kelvin?

Composition of Isomorphisms

Again no! Because we can compose isomorphisms:

```
fun <A,B,C> composeIso(iso1:Iso<A,B>, iso2: Iso<B,C>):
     Iso<A,C>{
  return Iso(
      get = { a ->
        iso2.get(iso1.get(a))},
      reverseGet = {c ->
        iso1.reverseGet(iso2.reverseGet(c))}
  )
}
```

Later we will use the general composition function coming from the Arrow-Optics library so this is here only for instructional value.

Composition of Isomorphisms

Let us finalize the part on isomorphisms with an example which uses inverse and composition:

```
// Defined with Fahrenheit in mind
fun increaseFahrenheit(fahrenheit: Fahrenheit):
    Fahrenheit{
  return Fahrenheit(fahrenheit.rawValue + 10)
}

// Used with Kelvin
val temperatureInKelvin = Kelvin(200.0)
val fahrenheitIsoKelvin =
  composeIso(inverseIso(celsiusIsoFahrenheit),
      celsiusIsoKelvin)
val increasedKelvin =
  asIf(fahrenheitIsoKelvin){c ->
    increaseFahrenheit(c)}(temperatureInKelvin)
```

Interlude: Arithmetic on Types

And now for something completely different. . .

Interlude: Arithmetic on Types

And now for something completely different...

There is a well known analogy between numbers with arithmetic operations and types with type constructors. This analogy allows us to transfer our intuition on numbers to types and reason about them more easily. Here is the gist of the idea for Kotlin types expressed as a table:

$a = b$	$a + b$	$a \times b$	a^b
`Iso<A,B>`	`Either<A,B>`	`Pair<A,B>`	`(B) -> A`

(In these notes we will mostly work with $+$, $\times$ and $=$.)

Interlude: Arithmetic on Types

And now for something completely different...

There is a well known analogy between numbers with arithmetic operations and types with type constructors. This analogy allows us to transfer our intuition on numbers to types and reason about them more easily. Here is the gist of the idea for Kotlin types expressed as a table:

$a = b$	$a + b$	$a \times b$	a^b
`Iso<A,B>`	`Either<A,B>`	`Pair<A,B>`	`(B) -> A`

(In these notes we will mostly work with $+$, $\times$ and $=$.)

Slogan: *Identities about numbers lift to isomorphisms about types!*

Interlude: Arithmetic on Types

Let us look at a few examples. We know tat $a + b = b + a$ for all numbers a and b. Therefore, by the previous slide, we should be able to construct an isomorphism of type

```
Iso<Either<A,B>,Either<B,A>>
```

for *fixed* types `A` and `B`. In Kotlin it is not possible to implement a polymorphic isomorphism directly because Kotlin has generics in functions but not in values. Of course this does not really matter as we are just building a mental model and the code we write here will not be used.

Interlude: Arithmetic on Types

Let us look at a few examples. We know tat $a + b = b + a$ for all numbers a and b. Therefore, by the previous slide, we should be able to construct an isomorphism of type

```
Iso<Either<A,B>,Either<B,A>>
```

for *fixed* types `A` and `B`. In Kotlin it is not possible to implement a polymorphic isomorphism directly because Kotlin has generics in functions but not in values. Of course this does not really matter as we are just building a mental model and the code we write here will not be used.

Nevertheless, we will include an indirect implementation for the sake of completeness. Feel free to ignore the details.

Interlude: Arithmetic on Types

The trick to encode generic values uses the following observation: There is a correspondence between values and functions whose domains have only one element. (Note that this is a lifting of the arithmetic identity $a^1 = a$.)

Interlude: Arithmetic on Types

The trick to encode generic values uses the following observation: There is a correspondence between values and functions whose domains have only one element. (Note that this is a lifting of the arithmetic identity $a^1 = a$.)

So let us start by creating a domain with a single element.

```
enum class One {ONE}
```

Interlude: Arithmetic on Types

The trick to encode generic values uses the following observation: There is a correspondence between values and functions whose domains have only one element. (Note that this is a lifting of the arithmetic identity $a^1 = a$.)

So let us start by creating a domain with a single element.

```
enum class One {ONE}
```

We have, say, for `Int`

```
val intAsFun: Iso<Int, (One) -> Int> = Iso(
  get = { n -> {_ -> n}},
  reverseGet = { f -> f(One.ONE)}
)
```

Now let us use this idea to show that `Either` is symmetric just like addition.

Interlude: Arithmetic on Types

Here is the implementation

```
fun <A,B> eitherSymmetry(@Suppress("UNUSED_PARAMETER")
    unused: One):
  Iso<Either<A,B>,Either<B,A>> {
return Iso(
  get = { p -> when(p) {
      is Either.Left -> Either.Right(p.value)
      is Either.Right -> Either.Left(p.value)
  }},
  reverseGet = { p -> when(p) {
      is Either.Left -> Either.Right(p.value)
      is Either.Right -> Either.Left(p.value)
  }}
)}
```

Interlude: Arithmetic on Types

Here is the implementation

```
fun <A,B> eitherSymmetry(@Suppress("UNUSED_PARAMETER")
    unused: One):
  Iso<Either<A,B>,Either<B,A>> {
return Iso(
  get = { p -> when(p) {
      is Either.Left -> Either.Right(p.value)
      is Either.Right -> Either.Left(p.value)
  }},
  reverseGet = { p -> when(p) {
      is Either.Left -> Either.Right(p.value)
      is Either.Right -> Either.Left(p.value)
  }}
)}
```

This is how it looks at a call site:

```
val swapped: Either<Int, String> =
  eitherSymmetry<String, Int>(One.ONE)
    .get(Either.Left("string"))
```

Interlude: Arithmetic on Types

Here is an example about multiplication: $(a \times b) \times c = a \times (b \times c)$ for all numbers a, b and c.

```
fun <A,B,C> timesAssoc(@Suppress("UNUSED_PARAMETER")
     unused: One):
   Iso<Pair<Pair<A,B>,C>,Pair<A,Pair<B,C>>> {
return Iso(
   get = { p ->
         Pair(p.first.first,
              Pair(p.first.second,p.second))},
   reverseGet = { p ->
         Pair(Pair(p.first, p.second.first),
              p.second.second)}
)}
```

Interlude: Arithmetic on Types

Final example: $a \times (b + c) = a \times b + a \times c$.

```
fun <A,B,C> distribute(@Suppress("UNUSED_PARAMETER")
    unused: One):
  Iso<Pair<A,Either<B,C>>,Either<Pair<A,B>,Pair<A,C>>>
        {
    return Iso(
      get = { p -> when(val sum = p.second){
        is Either.Left ->
             Either.Left(Pair(p.first, sum.value))
        is Either.Right ->
             Either.Right(Pair(p.first, sum.value))
      }},
      reverseGet = { sum -> when(sum) {
        is Either.Left ->
             Pair(sum.value.first,
                  Either.Left(sum.value.second))
        is Either.Right ->
             Pair(sum.value.first,
                  Either.Right(sum.value.second))
      }}
)}
```

Interlude: Arithmetic on Types

Let us summarize what we learned: There is a strong analogy between types and numbers but it is painful/awkward to express in Kotlin syntax.

Interlude: Arithmetic on Types

Let us summarize what we learned: There is a strong analogy between types and numbers but it is painful/awkward to express in Kotlin syntax.

Idea: Do not use Kotlin syntax! From now on we will denote `Either` and `Pair` by $+$ and $\times$, respectively. Since $=$ already has a meaning for types, we will denote `Iso` by $\cong$. This will make our reasoning much easier to follow and the ideas will be language independent.

Interlude: Arithmetic on Types

Let us summarize what we learned: There is a strong analogy between types and numbers but it is painful/awkward to express in Kotlin syntax.

Idea: Do not use Kotlin syntax! From now on we will denote `Either` and `Pair` by $+$ and $\times$, respectively. Since $=$ already has a meaning for types, we will denote `Iso` by $\cong$. This will make our reasoning much easier to follow and the ideas will be language independent.

Now back to optics –unless there are any questions.

Lenses Abstractly

A lens is a decomposition of a type into a product in which you are allowed to refer to only one component.

Lenses Abstractly

A lens is a decomposition of a type into a product in which you are allowed to refer to only one component.

More precisely, a lens from a type A to a type B is an isomorphism

$$\mathrm{Lens}(A, B) = A \cong B \times C$$

for some opaque (or hidden, or existential,...) type C.

Lenses Abstractly

A lens is a decomposition of a type into a product in which you are allowed to refer to only one component.

More precisely, a lens from a type A to a type B is an isomorphism

$$\mathrm{Lens}(A, B) = A \cong B \times C$$

for some opaque (or hidden, or existential,...) type C. In Kotlin we can express this definition as follows:

```
typealias ExistentialLens<A,B> = Iso<A,Pair<B,*>>
```

Note that this is a pedagogical model meant to be used when reasoning about lenses, *not* the actual implementation of lenses in Arrow-Optics.

Traditional Lenses

Here is a more traditional definition for lenses which is the basis of the lens implementation in Arrow-Optics. A lens from a type A to a type B consists of two functions $\mathrm{get}\colon A \to B$ and $\mathrm{set}\colon A \to B \to A$ satisfying the following laws:

Traditional Lenses

Here is a more traditional definition for lenses which is the basis of the lens implementation in Arrow-Optics. A lens from a type A to a type B consists of two functions $\mathrm{get}\colon A \to B$ and $\mathrm{set}\colon A \to B \to A$ satisfying the following laws:

- $\mathrm{get}(\mathrm{set}\, a\, b) = b$,

Traditional Lenses

Here is a more traditional definition for lenses which is the basis of the lens implementation in Arrow-Optics. A lens from a type A to a type B consists of two functions $\mathrm{get}\colon A \to B$ and $\mathrm{set}\colon A \to B \to A$ satisfying the following laws:

- $\mathrm{get}(\mathrm{set}\, a\, b) = b$,
- $\mathrm{set}\, a\, (\mathrm{get}\, a) = a$,

Traditional Lenses

Here is a more traditional definition for lenses which is the basis of the lens implementation in Arrow-Optics. A lens from a type A to a type B consists of two functions $\mathrm{get}\colon A \to B$ and $\mathrm{set}\colon A \to B \to A$ satisfying the following laws:

- $\mathrm{get}(\mathrm{set}\, a\, b) = b$,
- $\mathrm{set}\, a\, (\mathrm{get}\, a) = a$,
- $\mathrm{set}\, (\mathrm{set}\, a\, b_1)\, b_2 = \mathrm{set}\, a\, b_2$.

Traditional Lenses

Here is a more traditional definition for lenses which is the basis of the lens implementation in Arrow-Optics. A lens from a type A to a type B consists of two functions $\mathrm{get}\colon A \to B$ and $\mathrm{set}\colon A \to B \to A$ satisfying the following laws:

- $\mathrm{get}(\mathrm{set}\, a\, b) = b$,
- $\mathrm{set}\, a\, (\mathrm{get}\, a) = a$,
- $\mathrm{set}\, (\mathrm{set}\, a\, b_1)\, b_2 = \mathrm{set}\, a\, b_2$.

Note that modify can be implemented in terms of set and get so we do not need it here.
The typical examples are field access and update operations for records in most languages and views of database tables –with some caveats.

Traditional Lenses

Here is a more traditional definition for lenses which is the basis of the lens implementation in Arrow-Optics. A lens from a type A to a type B consists of two functions $\mathrm{get}\colon A \to B$ and $\mathrm{set}\colon A \to B \to A$ satisfying the following laws:

- $\mathrm{get}(\mathrm{set}\, a\, b) = b$,
- $\mathrm{set}\, a\, (\mathrm{get}\, a) = a$,
- $\mathrm{set}\, (\mathrm{set}\, a\, b_1)\, b_2 = \mathrm{set}\, a\, b_2$.

Note that modify can be implemented in terms of set and get so we do not need it here.
The typical examples are field access and update operations for records in most languages and views of database tables –with some caveats.
A particular example of interest for us is the pair type in Kotlin with the component access and update operations.

From Existential to Traditional

So what does this tell us about existential lenses? Well, we already have a traditional lens structure on pairs. So anything isomorphic to a pair can be made into a lens by transporting the lens structure through the isomorphism.

From Existential to Traditional

So what does this tell us about existential lenses? Well, we already have a traditional lens structure on pairs. So anything isomorphic to a pair can be made into a lens by transporting the lens structure through the isomorphism. Here is an implementation:

```
typealias ExistentialLens<A,B> = Iso<A,Pair<B,*>>

fun <A,B> fromExistential(eLens: ExistentialLens<A,B>)
    : Lens<A,B> {
  return Lens(
    get = { a ->
      eLens.get(a).first },
    set = { a, b ->
      eLens.reverseGet(Pair(b, eLens.get(a).second)) }
  )
}
```

From Traditional to Existential

How about the other direction? We can implement that direction, too, but the $*$ projection forces us to use an unchecked cast. Also there is no canonical choice for the second component of the pair so the construction takes another parameter from B to pick one.

From Traditional to Existential

How about the other direction? We can implement that direction, too, but the $*$ projection forces us to use an unchecked cast. Also there is no canonical choice for the second component of the pair so the construction takes another parameter from B to pick one.

Here is the implementation:

```
fun <A,B> toExistential(lens: Lens<A,B>, b: B):
    ExistentialLens<A,B> {
  return Iso(
    get = { a ->
      Pair(lens.get(a), lens.set(a, b)) },
    reverseGet = { p ->
      lens.set(p.second as A, p.first) } //Fishy!
  )
}
```

If this looks strange, don't worry. I included the function for the sake of completeness. We will not really use it.

Summary

After omitting a fair amount of detail we showed that there is only one lens up to isomorphism, and that is the natural lens structure on the pair type. This characterization is conceptually illuminating but awkward to code with. So it is better to work with the traditional lens implementation.

Summary

After omitting a fair amount of detail we showed that there is only one lens up to isomorphism, and that is the natural lens structure on the pair type. This characterization is conceptually illuminating but awkward to code with. So it is better to work with the traditional lens implementation.

Motto: Reason in terms of existential lenses, code using the traditional implementation.

Examples of Lenses

Now let us put the existential characterization we obtained to use. Consider the following data classes:

```
data class Location (
  val latitude: Float,
  val longitude: Float
)

data class Weather(
  val temperature: Celsius,
  val date: Date,
  val location: Location
)
```

Examples of Lenses

The Weather type naturally is a product:

$$\texttt{Weather} \cong \texttt{Celsius} \times \texttt{Date} \times \texttt{Location}$$

Examples of Lenses

The Weather type naturally is a product:

$$\texttt{Weather} \cong \texttt{Celsius} \times \texttt{Date} \times \texttt{Location}$$

So singling out any of the fields gives a lens:

$$\begin{aligned}\texttt{Weather} &\cong \texttt{Celsius} \times (\texttt{Date} \times \texttt{Location}) \\ &\cong \texttt{Date} \times (\texttt{Celsius} \times \texttt{Location}) \\ &\cong \texttt{Location} \times (\texttt{Date} \times \texttt{Celsius})\end{aligned}$$

Each of these decompositions gives rise to a lens.

Examples of Lenses

One can also single out a group of fields. For instance the decomposition

$$\texttt{Weather} \cong (\texttt{Celsius} \times \texttt{Date}) \times \texttt{Location}$$

gives a lens of type `Lens<Weather,Pair<Celsius,Date>>`. It accesses/updates the fields `temperature` and `date` simultaneously.

Examples of Lenses

Now let's do something slightly more interesting. We know that

$$\texttt{Fahrenheit} \cong \texttt{Celsius}$$

This gives us

$$\begin{aligned}\texttt{Weather} &\cong \texttt{Celsius} \times (\texttt{Date} \times \texttt{Location}) \\ &\cong \texttt{Fahrenheit} \times (\texttt{Date} \times \texttt{Location})\end{aligned}$$

Therefore we should have a lens of type `Lens<Weather,Fahrenheit>`

Examples of Lenses

Now let's do something slightly more interesting. We know that

$$\texttt{Fahrenheit} \cong \texttt{Celsius}$$

This gives us

$$\begin{aligned}\texttt{Weather} &\cong \texttt{Celsius} \times (\texttt{Date} \times \texttt{Location}) \\ &\cong \texttt{Fahrenheit} \times (\texttt{Date} \times \texttt{Location})\end{aligned}$$

Therefore we should have a lens of type `Lens<Weather,Fahrenheit>`

Of course the more general observation is that the composition of a lens with an isomorphism is again a lens. Moreover, this composition is supported by the Arrow-Optics library!

```
val weatherFahrenheit =
  weatherCelsius compose celsiusIsoFahrenheit
```

The lens `weatherFahrenheit` is sometimes called a virtual field.

Examples of Lenses

Another example: Note that

$$\texttt{Location} \cong \texttt{Float}_{\mathit{lat}} \times \texttt{Float}_{\mathit{long}}$$

Inlining this in `Weather` gives us

$$\begin{aligned}\texttt{Weather} &\cong \texttt{Celsius} \times \texttt{Date} \times \texttt{Location} \\ &\cong \texttt{Celsius} \times \texttt{Date} \times (\texttt{Float}_{\mathit{lat}} \times \texttt{Float}_{\mathit{long}}) \\ &\cong \texttt{Float}_{\mathit{lat}} \times (\texttt{Celsius} \times \texttt{Date} \times \texttt{Float}_{\mathit{long}})\end{aligned}$$

Therefore we should have a lens of type `Lens<Weather,Float>` which focuses on the latitude.

Examples of Lenses

Another example: Note that

$$\texttt{Location} \cong \texttt{Float}_{lat} \times \texttt{Float}_{long}$$

Inlining this in `Weather` gives us

$$\begin{aligned}\texttt{Weather} &\cong \texttt{Celsius} \times \texttt{Date} \times \texttt{Location} \\ &\cong \texttt{Celsius} \times \texttt{Date} \times (\texttt{Float}_{lat} \times \texttt{Float}_{long}) \\ &\cong \texttt{Float}_{lat} \times (\texttt{Celsius} \times \texttt{Date} \times \texttt{Float}_{long})\end{aligned}$$

Therefore we should have a lens of type `Lens<Weather,Float>` which focuses on the latitude.

Again there is a more general observation to be made: composition of lenses is a lens. This composition is also supported by the Arrow-Optics library

```
val weatherLatitude =
  weatherLocation compose locationLatitude
```

Examples of Lenses

Let us finish the part on lenses with an example which is not as well known as it should be. Consider the following types:

```
data class Weight1(val net: Float, val tare: Float)
data class Weight2(val gross: Float, val tare: Float)
```

and the specific isomorphism between them:

```
val weight1IsoWeight2: Iso<Weight1,Weight2> = Iso(
  get = { weight1 ->
    Weight2(gross = weight1.net + weight1.tare,
            tare = weight1.tare) },
  reverseGet = { weight2 ->
    Weight1(net = weight2.gross - weight2.tare,
            tare = weight2.tare) }
)
```

Examples of Lenses

These isomorphisms give us

$$\texttt{Weight1} \cong \texttt{Weight2} \cong \texttt{Float}_{\mathit{gross}} \times \texttt{Float}_{\mathit{tare}}$$

Therefore we must have a lens of type `Lens<Weight1,Float>` focusing on gross weight even though `Weight1` has no field for gross weight.

Examples of Lenses

These isomorphisms give us

$$\texttt{Weight1} \cong \texttt{Weight2} \cong \texttt{Float}_{\mathit{gross}} \times \texttt{Float}_{\mathit{tare}}$$

Therefore we must have a lens of type `Lens<Weight1,Float>` focusing on gross weight even though `Weight1` has no field for gross weight.

Arrow-Optics library allows us to express this lens as

```
val weight1Gross: Lens<Weight1,Float> =
  weight1IsoWeight2 compose weight2gross
```

where `weight2gross: Lens<Weight2,Float>` is the natural lens focusing on gross weight.

Examples of Lenses

These isomorphisms give us

$$\texttt{Weight1} \cong \texttt{Weight2} \cong \texttt{Float}_{gross} \times \texttt{Float}_{tare}$$

Therefore we must have a lens of type `Lens<Weight1,Float>` focusing on gross weight even though `Weight1` has no field for gross weight.

Arrow-Optics library allows us to express this lens as

```
val weight1Gross: Lens<Weight1,Float> =
  weight1IsoWeight2 compose weight2gross
```

where `weight2gross: Lens<Weight2,Float>` is the natural lens focusing on gross weight.

The general observation is that the composition of an isomorphism with a lens is again a lens.

Examples of Lenses

In case you want to see, this is the same lens implemented manually:

```
val weight1GrossHandmade: Lens<Weight1,Float> = Lens(
  get = { weight1 ->
    weight1.net + weight1.tare },
  set = { weight1, newGross ->
    Weight1(net = newGross - weight1.tare,
            tare = weight1.tare) }
)
```

Prisms Abstractly

A prism is a decomposition of a type into a sum in which you are allowed to refer to only one side.

Prisms Abstractly

A prism is a decomposition of a type into a sum in which you are allowed to refer to only one side.

More precisely, a prism from a type A to a type B is an isomorphisms

$$\mathrm{Prism}(A, B) = A \cong B + C$$

for some opaque (or hidden, or existential,...) type C. In Kotlin we can express this definition as follows:

```kotlin
typealias ExistentialLens<A,B> = Iso<A,Either<B,*>>
```

Again, this is not the actual implementation. You know the drill...

Traditional Prisms

As in the case of lenses, there is a traditional law based definition of prisms. A prism from type A to type B consists of two functions $\mathrm{review}\colon B \to A$ and $\mathrm{preview}\colon A \to \mathrm{Option}\, B$ satisfying the following laws:

Traditional Prisms

As in the case of lenses, there is a traditional law based definition of prisms. A prism from type A to type B consists of two functions $\mathrm{review}\colon B \to A$ and $\mathrm{preview}\colon A \to \mathrm{Option}\, B$ satisfying the following laws:

- $\mathrm{preview}(\mathrm{review}\, b) = \mathrm{Some}\, b$

Traditional Prisms

As in the case of lenses, there is a traditional law based definition of prisms. A prism from type A to type B consists of two functions $\mathrm{review} \colon B \to A$ and $\mathrm{preview} \colon A \to \mathrm{Option}\, B$ satisfying the following laws:

- $\mathrm{preview}(\mathrm{review}\, b) = \mathrm{Some}\, b$
- if $\mathrm{preview}\, a = \mathrm{Some}\, b$ then $\mathrm{review}\, b = a$.

Traditional Prisms

As in the case of lenses, there is a traditional law based definition of prisms. A prism from type A to type B consists of two functions $\mathrm{review}\colon B \to A$ and $\mathrm{preview}\colon A \to \mathrm{Option}\, B$ satisfying the following laws:

- $\mathrm{preview}(\mathrm{review}\, b) = \mathrm{Some}\, b$
- if $\mathrm{preview}\, a = \mathrm{Some}\, b$ then $\mathrm{review}\, b = a$.

Note that a prism can be thought of as a partial isomorphism in the sense that $\mathrm{preview}$ and review are almost inverses of each other.

Traditional Prisms

The implementation in Kotlin uses the same functions with different names: review is called `reverseGet` and $\mathrm{preview}$ is called `getOption`.

Traditional Prisms

The implementation in Kotlin uses the same functions with different names: review is called `reverseGet` and preview is called `getOption`.

Also, instead of `getOption` you may see a function called `getOrModify` of type `(A) -> Either<A,B>` in documentation –so `Option<B>` is replaced by `Either<A,B>`. This function returns the original input from `A` if the `getOption` function produces `None`.

Traditional Prisms

We can again implement the connection between these two formulations in Kotlin with some caveats.

```
fun <A,B> fromExistentialPrism(ePrism:
    ExistentialPrism<A,B>): Prism<A, B> {
  return Prism(
    getOption = { a ->
      when(val eb = ePrism.get(a)){
        is Either.Left -> Some(eb.value)
        is Either.Right -> None
      }},
      reverseGet = { b ->
        ePrism.reverseGet(Either.Left(b)) }
)}
```

Traditional Prisms

And here is the other direction.

```
fun <A,B> toExistentialPrism(prism: Prism<A,B>):
    ExistentialPrism<A,B>{
  return Iso(
    get = { a -> when(val eb = prism.getOrModify(a)){
      is Either.Left -> Either.Right(eb.value)
      is Either.Right -> Either.Left(eb.value)
    }},
    reverseGet = { eb -> when(eb){
      is Either.Left -> prism.reverseGet(eb.value)
      is Either.Right -> eb.value as A // Fishy!
    }}
)}
```

Examples of Prisms

As in the case of lenses we immediately see that prisms are closed under composition. Also composition of a lense with an isomorphism from both sides is again a prism. I leave the details to you.

Examples of Prisms

Let us start with a generic example. For enum classes, you only need `reverseGet` to define a prism.

```
// Assumes revGet is injective
inline fun <A, reified B: Enum<B>>
    makePrismFromHalf(noinline revGet: (B) -> A):
        Prism<A,B>{
  val mkPair = { b:B -> Pair(revGet(b), b) }
  val lookupTable: Map<A,B> =
        enumValues<B>().asList().associate(mkPair)
  return Prism(
      getOption = { a -> lookupTable.getOrNone(a) },
      reverseGet = revGet
  )
```

Injectivity assumption in the comment means that `revGet` sends distinct inputs to distinct outputs.

Examples of Prisms

Let us look at a concrete example.

```
enum class Days(val prettyName: String)
  { MONDAY("Monday"),
    TUESDAY("Tuesday"),
    WEDNESDAY("Wednesday"),
    THURSDAY("Thursday"),
    FRIDAY("Friday"),
    SATURDAY("Saturday"),
    SUNDAY("Sunday")
  }
```

Clearly `prettyName` is injective, so we have a prism:

```
val dayParser: Prism<String,Days> =
  makePrismFromHalf { d -> d.prettyName }
```

Not too exciting...

Examples of Prisms

Now let us build on this example a little.

```
enum class WorkDays
  { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY }

val daysPrismWorkDays: Prism<Days,WorkDays> =
  makePrismFromHalf { wd ->
    Days.values()[wd.ordinal] }
    //Possible ArrayIndexOutOfBoundsException

val weekDayParser: Prism<String,WorkDays> =
  dayParser compose daysPrismWorkDays
```

Examples of Prisms

Now let us build on this example a little.

```
enum class WorkDays
  { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY }

val daysPrismWorkDays: Prism<Days,WorkDays> =
  makePrismFromHalf { wd ->
    Days.values()[wd.ordinal] }
    //Possible ArrayIndexOutOfBoundsException

val weekDayParser: Prism<String,WorkDays> =
  dayParser compose daysPrismWorkDays
```

We get a parser/pretty-printer for week days for free! Also, the source of pretty names for weekend days and work days is the same and is unique. If we want to change it we we only change `prettyName` in the `Days` enum. So design is more composable.

Examples of Prisms

Now let us build on this example a little.

```
enum class WorkDays
  { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY }

val daysPrismWorkDays: Prism<Days,WorkDays> =
  makePrismFromHalf { wd ->
    Days.values()[wd.ordinal] }
    //Possible ArrayIndexOutOfBoundsException

val weekDayParser: Prism<String,WorkDays> =
  dayParser compose daysPrismWorkDays
```

We get a parser/pretty-printer for week days for free! Also, the source of pretty names for weekend days and work days is the same and is unique. If we want to change it we we only change `prettyName` in the `Days` enum. So design is more composable.

One final advantage is that it is really easy to write tests since there are test functions for prisms –actually for all optics– in Arrow-Optics.

Appendix I: Isomorphisms as Other Optics

Recall the following enum

```
enum class One {ONE}
```

If we denote this type by 1 then we have the following isomorphism: $X \cong X \times 1$. Thus, if we have an isomorphism $A \cong B$, then we also have $A \cong B \times 1$. Therefore every isomorphism is a lens.

Appendix I: Isomorphisms as Other Optics

Recall the following enum

```
enum class One {ONE}
```

If we denote this type by 1 then we have the following isomorphism: $X \cong X \times 1$. Thus, if we have an isomorphism $A \cong B$, then we also have $A \cong B \times 1$. Therefore every isomorphism is a lens.

Similarly using the class

```
enum class Zero {}
```

and the identity $X \cong X + 0$ we also deduce every isomorphism is also a prism.

Appendix I: Isomorphisms as Other Optics

Recall the following enum

```
enum class One {ONE}
```

If we denote this type by 1 then we have the following isomorphism: $X \cong X \times 1$. Thus, if we have an isomorphism $A \cong B$, then we also have $A \cong B \times 1$. Therefore every isomorphism is a lens.

Similarly using the class

```
enum class Zero {}
```

and the identity $X \cong X + 0$ we also deduce every isomorphism is also a prism.

These observations are supported by Arrow-Optics, so we can use `set` and `getOrModify` functions on isomorphisms.

Appendix II: Affine Traversals

An affine traversal is what kotlin calls an `Optional`. An Optional from A to B is an isomorphism

$$\mathrm{Optional}(A, B) = A \cong B \times C + D$$

for some opaque types C and D.

Appendix II: Affine Traversals

An affine traversal is what kotlin calls an `Optional`. An Optional from A to B is an isomorphism

$$\mathrm{Optional}(A, B) = A \cong B \times C + D$$

for some opaque types C and D.

They usually arise as compositions of lenses and prisms. Suppose we have a lens $A \cong B \times C$ and a prism $B \cong D + E$. Then inlining B gives

$$A \cong B \times C \cong (D + E) \times C \cong D \times C + E \times C$$

It is easy to show that composition in the other way around also gives an optional.

Appendix III: Setters

Setters also have a nice existential characterization. A setter from A to B is an isomorphism

$$\mathrm{Setter}(A, B) = A \cong F(B)$$

where F is an opaque “mappable” type constructor. The technical term is functor and Arrow used to have functors but in the upcoming version it seems to be dropped so I will not say anything more on the subject.

Thank you for listening!