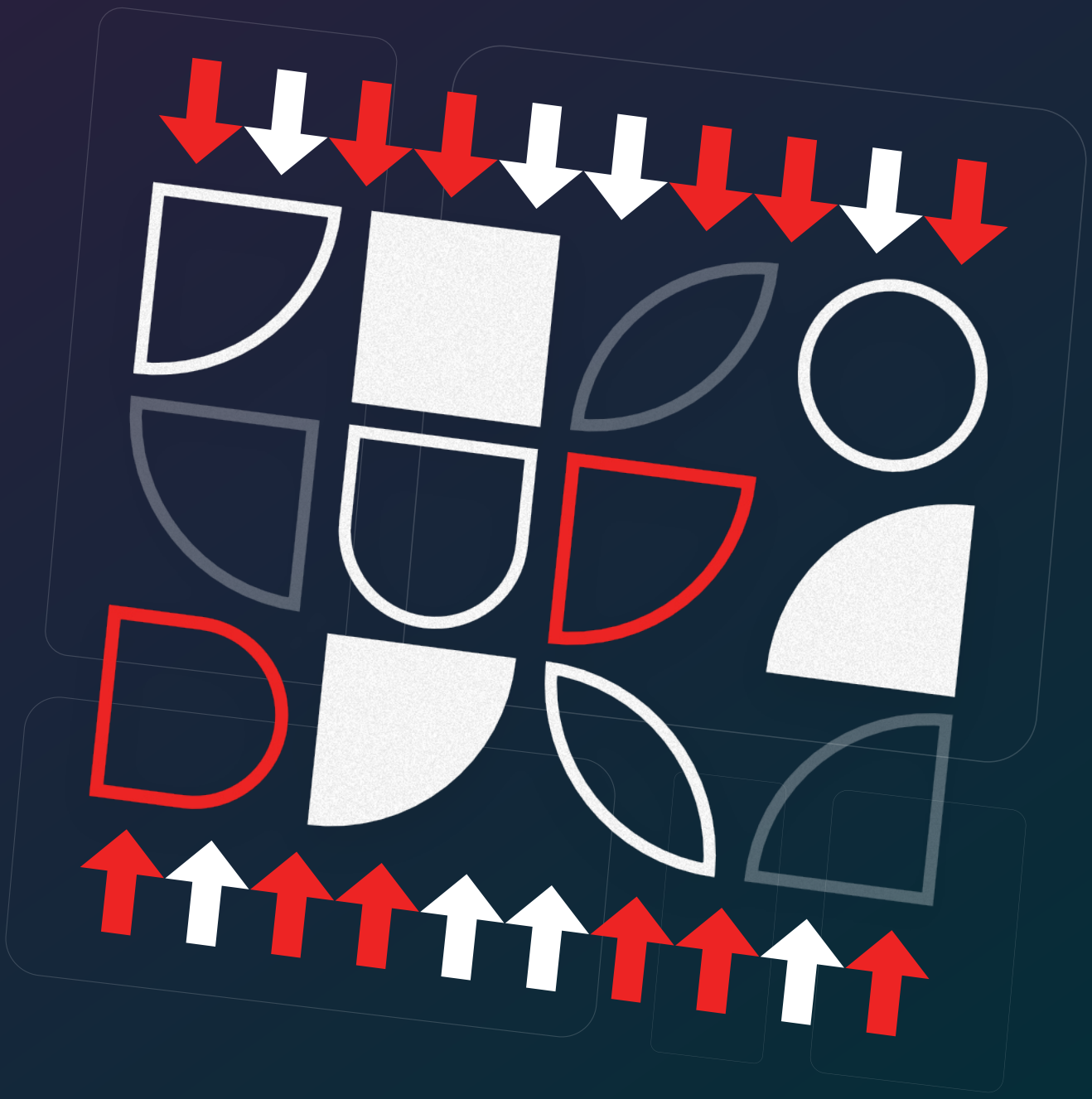
scalac

JORGE VASQUEZ

IMPROVE YOUR FOCUS WITH ZIO OPTICS



Introduction

Functional Programming (FP) is a great programming paradigm. This is because programming with pure functions gives developers several advantages, such as code that is concise, predictable, and easier to test. However, it turns out there are some cases where code written using the Functional paradigm is more verbose than with the Object-Oriented paradigm (OOP). This happens especially because in Functional Programming we must work with immutable data structures. For example, when we have a structure with deeply nested records, complications start to arise when trying to modify a part of it. That's because we can't actually modify the original structure, so we need to create a completely new one that's almost exactly the same as the original. However, it will need a little modification, and there's usually a lot of boilerplate code we will have to write to solve that problem. In OOP, we can just change the value we want, and that's it. However, when we start working with deeply nested collections, not even OOP has a good solution to avoid boilerplate. The good news is that in FP we have a tool at our disposal for addressing these problems. It's called Optics.

In the Scala world, there are some libraries which offer implementations of optics. In this document, we are going to focus on ZIO Optics, which is a new library in the ZIO ecosystem.

Here is what you will learn:

- Why Optics matter and what problems they can help solve in FP.
- Why ZIO Optics matters.
- The `Optic` data type and its different flavors: `Lens`, `Prism`, `Iso`, `Optional` and `Traversal`.
- Polymorphic optics and why they are important.
- How to use the full power of optics by composing them.
- Effectful and Transactional Optics.

Overview of ZIO Optics

Why Optics matter

Optics are a very important tool in FP. They allow developers to easily access and manipulate deeply nested immutable data structures.

For example, let's say we have a Person case class, which contains an Address case class inside , which in turn contains a Street case class inside:

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

If we want to change the number of the street where a Person lives, just using plain Scala with no optics, we will need to do something like this:

```
def setStreetNumber(person: Person, newStreetNumber: Int): Person =
  person.copy(
    address = person.address.copy(
      street = person.address.street.copy(
        number = newStreetNumber
      )
    )
  )
```

That's a lot of ceremony, and if we had defined Person as a normal class instead of a case class, it would be even worse, because we wouldn't have the copy method at our disposal.

But, if we were doing OOP, Person would be a mutable data structure:

```
final class MutablePerson(var fullName: String, var address: MutableAddress)
final class MutableAddress(var city: String, var street: MutableStreet)
final class MutableStreet(var name: String, var number: Int)
```

That means, setting a new street number would now be a piece of cake:

```
def setStreetNumberMutable(person: MutablePerson, newStreetNumber: Int): Unit =
  person.address.street.number = newStreetNumber
```

Here is where OOP really shines: mutating deeply nested records. That's why we'd like to have something similar in FP, which is precisely where optics steps in. We'll be seeing how in the following sections, so stay tuned!

Having said that, there are other cases that are really hard to handle in OOP. For example, in OOP it's typical to work with `null` values, right? In that case, the `setStreetNumberMutable` method gets a lot more complicated:

```
def setStreetNumberMutableNull(person: MutablePerson, newStreetNumber: Int): Unit =
  if (person != null) {
    if (person.address != null) {
      if (person.address.street != null) {
        person.address.street.number = newStreetNumber
      } else {
        throw new Exception("Null street!")
      }
    } else {
      throw new Exception("Null address!")
    }
  } else {
    throw new Exception("Null person!")
  }
```

On the other hand, in FP we don't work with `null` values, we use the `Option` data type instead. So, our data models would have to be like this:

```
final case class OPerson(fullName: Option[String], address: Option[OAddress])
final case class OAddress(city: Option[String], street: Option[OStreet])
final case class OStreet(name: Option[String], number: Option[Int])
```

So `setStreetNumber` would look like this:

```
def setStreetNumberOptional(person: OPerson, newStreetNumber: Int): Either[String, OPerson] =
  person.address match {
    case Some(address) =>
      address.street match {
        case Some(street) =>
          Right(
            OPerson(
              fullName = person.fullName,
              address = Some(OAddress(address.city, Some(OStreet(street.name, Some(newStreetNumber)))))
            )
          )
        case None => Left("Empty street")
      }
    case None => Left("Empty address")
  }
```

Lots of boilerplate, right? And as demonstrated before, OOP didn't have a good solution either for this scenario. However, in FP, optics could help a lot.

Finally, let's consider one more example. Let's say we have a `Map` of `Customers`, indexed by customer ID:

```
val customers: Map[Long, Customer] = Map(
  1000L -> Customer("Mary Lopez", Map(1L    -> Order(1, 100), 2L -> Order(2, 200))),
  2000L -> Customer("David Adams", Map(1L   -> Order(3, 300), 2L -> Order(4, 400))),
  3000L -> Customer("Brian Johnson", Map(1L -> Order(5, 500), 2L -> Order(6, 600)))
)
```

The basic data structures are:

```
final case class Order(itemId: Long, quantity: Long)
final case class Customer(name: String, orders: Map[Long, Order])
```

Let's implement a method to modify the quantity of an Order, given the customer ID and order ID, using FP:

```
def setQuantity(
 customers: Map[Long, Customer],
 customerId: Long,
 orderId: Long,
 newQuantity: Long
): Either[String, Map[Long, Customer]] =
  customers.get(customerId) match {
    case Some(customer) =>
      customer.orders.get(orderId) match {
        case Some(order) =>
          Right(
            customers.updated(
              customerId,
              customer.copy(
                orders = customer.orders.updated(
                  orderId,
                  order.copy(quantity = newQuantity)
                )
              )
            )
          )
        case None =>
          Left(s"Order with ID $orderId does not exist")
      }
    case None =>
      Left(s"Customer with ID $customerId does not exist")
  }
```

Again, lots of boilerplate. And if we tried to implement that solution with OOP, you can imagine it wouldn't be great either. Once more, optics would be a perfect solution to solve this problem in an elegant way.

Enter ZIO Optics

ZIO Optics is a new library in the ZIO ecosystem, which as its name implies provides an implementation of optics .

There are other optics libraries for Scala such as Monocle, which is really great, but ZIO Optics has some very important features. For example, other libraries provide just Pure Optics, meaning optics that only work with pure data living in memory. On the other hand, ZIO Optics provides not just that, but also Effectful Optics. These are very powerful because they can interact with the outside world. They do some IO to work with data that doesn't necessarily live in memory, but comes from external sources such as databases, files, APIs, etc. Not only that, ZIO Optics also provides Transactional Optics, which are able to participate in STM transactions. If you want to know more about ZIO STM, you can take a look at this article. Another very important differentiator of ZIO Optics is that it's designed to work in a concurrent, multi-threaded environment. Optics in other libraries are designed for a world of just one thread of execution.

Some other important characteristics of ZIO Optics are:

- It has a completely different encoding for optics, which enables superior composability. We'll be seeing more on this later, in the section on composition of optics.
- It offers clear error messages for easy diagnostics.
- Since it uses Scala modules from the very beginning, it's very extensible. This means, for example, you could easily extend ZIO Optics to integrate with other effect systems, such as Cats-Effect or Monix.
- It has full integration with the ZIO ecosystem.

How to use ZIO Optics in your project

If you want to use ZIO Optics in your project, you just need to add the following dependency to your `build.sbt` file:

```
libraryDependencies += "dev.zio" %% "zio-optics" % "0.1.0"
```

The Optic data type

The core data type in the ZIO Optics library is called `Optic`, and it is just a combination of two functions, a getter and a setter:

```
case class Optic[
  -GetWhole,
  -SetWholeBefore,
  -SetPiece,
  +GetError,
  +SetError,
  +GetPiece,
  +SetWholeAfter
](
  getOptic: GetWhole => OpticResult[(GetError, SetWholeAfter), GetPiece],
  setOptic: SetPiece => SetWholeBefore => OpticResult[(SetError, SetWholeAfter), SetWholeAfter]
)
```

You can see `Optic` has several type parameters, seven of them in all! The reason it has so many parameters is to enable maximum flexibility, so that all kinds of optics can be simply derived from this single data type.

From the [ZIO Optics documentation] we get an explanation of what the type parameters mean:

- The getter is just a function which can take some larger data structure (i.e. the whole) of type `GetWhole` and get a part of it (i.e. a piece) of type `GetPiece`. It can potentially fail with an error of type `GetError` because the piece we are trying to get might not exist in the whole.
- The setter is another function which has the ability, given some piece of type `SetPiece` and an original whole of type `SetWholeBefore`, to return a new structure of type `SetWholeAfter`. Setting can fail with an error of type `SetError` because the piece we are trying to set might not exist in the whole.

If you are a little confused about this, don't worry because we normally won't need to interact directly with this `Optic` data type. Instead, we are going to work with specific cases of Optic which are a lot easier to reason about. The idea of the `Optic` data type in ZIO Optics is that, if we play around with its type parameters, we can get different types of optics for free. We'll be exploring them in the next sections.

General categories of optics in ZIO Optics

In the previous section, you could see that both the getter and the setter of an `Optic` return a value of type `OpticResult[+E, +A]`. This represents the result of executing an `Optic` which can be either a success of type A or a failure of type E.

`OpticResult[E, A]` is just an abstract type that changes according to the `Optic` category, and ZIO Optics has three of them in general:

- **Pure Optics:** Where OpticResult[E, A] = Either[E, A], meaning these optics can just return pure values.
- **Effectful Optics:** Where OpticResult[E, A] = IO[E, A], meaning these optics can return ZIO effects, which opens a whole new space of possibilities. That's because it implies we can have optics that interact with the outside world and are not limited to working with data in memory.
- **Transactional Optics:** Where OpticResult[E, A] = STM[E, A], meaning these optics can participate in ZIO STM transactions.

Exploring Pure Optics

As mentioned above, Pure Optics are the ones where `OpticResult[E, A] = Either[E, A]`. So, we have:

```
case class Optic[
  -GetWhole,
  -SetWholeBefore,
  -SetPiece,
  +GetError,
  +SetError,
  +GetPiece,
  +SetWholeAfter
](
  getOptic: GetWhole => Either[(GetError, SetWholeAfter), GetPiece],
  setOptic: SetPiece => SetWholeBefore => Either[(SetError, SetWholeAfter), SetWholeAfter]
)
```

To work with Pure Optics, just include the following import:

```
import zio.optics._
```

Let's explore now the different kinds of Pure Optics we have at our disposal.

Lens

A `Lens` is an `Optic` that accesses a field of a product type, which is a specific category of so called Algebraic Data Types. If you don't know what ADTs are, I recommend reading this article from the Functional Programming Simplified book, by Alvin Alexander. Examples of product types are tuples or case classes. In ZIO Optics, a Lens is represented by the following type alias:

```
type Lens[S, A] = Optic[S, S, A, Nothing, Nothing, A, S]
```

So, the simplified signature of `Optic` for `Lens` would be like this:

```
case class Lens[S, A](
  getOptic: S => Either[Nothing, A],
  setOptic: A => S => Either[Nothing, S]
)
```

To work with Pure Optics, just include the following import:

- The `GetError` and `SetError` types of a `Lens` are `Nothing` because getting or setting a field of a product type cannot fail (the field will always be present).
- The `GetWhole`, `SetWholeBefore`, and `SetWholeAfter` types are the same and represent the product type `S` we are working with.
- The `GetPiece` and `SetPiece` types are also the same and represent the field `A` we are trying to access.

To have a cleaner mental model of a `Lens`, we can simplify its signature even further:

```
case class Lens[S, A](
  getOptic: S => Either[Nothing, A],
  setOptic: A => S => Either[Nothing, S]
)
```

This way, it's easier to realize that a `Lens` is an `Optic` with which we can always:

- Get a field `A` of a product type `S`.
- Set a new value for a field of a product type `S`, given a new value `A` and the original structure `S`.

Examples with Lens

Let's now see how to construct some `Lenses`. Imagine we have the following `Person` product type:

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

Because this `Person` type has two fields, we can create two `Lenses`. Let's first create a `Lens` for the name field, and let's put it inside the `Person` companion object:

```
import zio.optics._

object Person {
  val name: Lens[Person, String] =
    Lens(
      person => Right(person.name),
```

```
      newName => person => Right(person.copy(name = newName))
    )
}
```

Note that, to create a `Lens`, we just need to call the `Lens.apply` method providing a getter and a setter for the field we want to access (in this case: `name`).

And, creating a `Lens` for the age field is very similar:

```
mport zio.optics._

object Person {
  ...

  val age: Lens[Person, Int] =
    Lens(
      person => Right(person.age),
      newAge => person => Right(person.copy(age = newAge))
    )
}
```

We can now use these new `Lenses` we've just defined, to get and set values of any `Person` object:

```
val person1 = Person("Juanito", 25)

// Get `name` from person1
val name1: Either[Nothing, String] = Person.name.get(person1)
// Right(Juanito)

// Get `age` from person1
val age1: Either[Nothing, Int] = Person.age.get(person1)
// Right(25)



// Change `name` in person1 to Pepito
val person2: Either[Nothing, Person] = Person.name.set("Pepito")(person1)
// Right(Person(Pepito, 25))

// Change `age` in person1 to 27
val person3: Either[Nothing, Person] = Person.age.set(27)(person1)
// Right(Person(Juanito, 27))

// Increase `age` of person1 by 5
val person4: Either[Nothing, Person] = Person.age.update(person1)(_ + 5)
// Right(Person(Juanito, 30))
```

Finally, ZIO Optics includes some built-in `Lenses`. For example:

```
val first  = Lens.first  // Get the first element of a Tuple2
val second = Lens.second // Get the second element of a Tuple2
```

Prism

A `Prism` is an `Optic` that accesses a case of a sum type (also called coproduct type), such as the `Left` or `Right` cases of an Either or one of the subtypes of a sealed trait. In ZIO `Optics`, a `Prism` is represented by the following type alias:

```
type Prism[S, A] = Optic[S, Any, A, OpticFailure, Nothing, A, S]
```

So, the simplified signature of `Optic` for `Prism` would be like this:

```
case class Prism[S, A](
  getOptic: S => Either[OpticFailure, A],
  setOptic: A => Any => Either[Nothing, S]
)
```

This means that:

- The `GetError` type of a Prism is `OpticFailure` because getting a case of a sum type can fail when the case we are trying to access does not exist. For example, we might be trying to access the right side of an Either but the instance we are working with is actually a Left. By the way, in ZIO Optics the `OpticFailure` type models the different ways getting or setting can fail.
- The `SetError` type is `Nothing` because given one of the cases of a sum type we can always return a new value of the aforementioned sum type.
- The `GetWhole` and `SetWholeAfter` types are the same and represent the sum type S.
- The `SetWholeBefore` type is set to Any because we don't need any original structure to set a new value. This is different from `Lens`, where we needed the original structure. A sum type consists of nothing but its cases so, if we have a new value of the case we want to set, we can just use that value and we don't need the original structure.

- The `GetPiece` and `SetPiece` types are the same and represent the case A we are trying to access.

To have a cleaner mental model of a `Prism`, we can simplify its signature even further:

```
case class Prism[S, A](
  getOptic: S => Either[OpticFailure, A],
  setOptic: A => S
)
```

This way, it's easier to realize that a `Prism` is an `Optic` with which we can:

- Get a case `A` of a sum type `S`. This can fail with an `OpticFailure`.
- Set a new value for a sum type `S`, given a new case `A`.

Examples with Prism

Suppose we have the following sum type for Json values:

```
sealed trait Json
object Json {
  final case object JNull                  extends Json
  final case class JStr(v: String)         extends Json
  final case class JNum(v: Double)         extends Json
  final case class JArr(v: Vector[Json])   extends Json
  final case class JObj(v: Map[String, Json]) extends Json

  type JNull = JNull.type
}
```

Because this `Json` type has five cases, we can create five Prisms. Let's first create a `Prism` for the `JNull` case, and let's put it inside the Json companion object:

```
import zio.optics._

object Json {
  ...

  val jNull: Prism[Json, JNull] =
    Prism(
      {
        case json: JNull => Right(json)
        case _           => Left(OpticFailure("Not a JNull"))
      },
```

```
      Right(_)
    )
}
```

So, to create a `Prism`, we just need to call the `Prism.apply` method providing a getter and a setter for the case we want to access (`JNull`). You can see that:

- The getter can fail with an `OpticFailure` when trying to get values that are not of type `JNull`.
- The setter succeeds right away with the provided `JNull`.

And, creating `Prisms` for other cases is very similar:

```
import zio.optics._

object Json {
  ...

  val jStr: Prism[Json, JStr] =
    Prism(
      {
        case json: JStr => Right(json)
        case _          => Left(OpticFailure("Not a JStr"))
      },
      Right(_)
    )

  val jNum: Prism[Json, JNum] =
    Prism(
      {
        case json: JNum => Right(json)
        case _          => Left(OpticFailure("Not a JNum"))
      },
      Right(_)
    )

  val jArr: Prism[Json, JArr] =
    Prism(
      {
        case json: JArr => Right(json)
        case _          => Left(OpticFailure("Not a JArr"))
      },
      Right(_)
    )


  val jObj: Prism[Json, JObj] =
    Prism(
      {
```

```
        case json: JObj => Right(json)
        case _          => Left(OpticFailure("Not a JObj"))
      },
      Right(_)
    )
}
```

We can now use these new `Prisms` we've just defined to get and set values of any `Json` object:

```
val json1 = Json.JNum(100)
val json2 = Json.JStr("hello")

// Get a JNum from json1
val jNum1: Either[OpticFailure, Json.JNum] = Json.jNum.get(json1)
// Right(JNum(100.0))

// Get a JNum from json2
val jNum2: Either[OpticFailure, Json.JNum] = Json.jNum.get(json2)
// Left(OpticFailure(Not a JNum))

// Set Json to JStr, notice we don't need the whole!
val json3: Either[Nothing, Json] = Json.jStr.set(Json.JStr("some text"))
// Right(JStr(some text))



// Update json1 duplicating its value
val json4: Either[OpticFailure, Json] = Json.jNum.update(json1) {
 case Json.JNum(x) => Json.JNum(x * 2)
}
// Right(JNum(200.0))

// Update json2 duplicating its value
val json5: Either[OpticFailure, Json] = Json.jNum.update(json2) {
 case Json.JNum(x) => Json.JNum(x * 2)
}
// Left(OpticFailure(Not a JNum))
```

And by the way, ZIO Optics includes some built-in `Prisms`, for example:

```
val cons  = Prism.cons  // Access the :: case of a List
val left  = Prism.left  // Access the Left case of an Either
val right = Prism.right // Access the Right case of an Either
val none  = Prism.none  // Access the None case of an Option
val some  = Prism.some  // Access the Some case of an Option
```

Iso

An `Iso` is an `Optic` that accesses a part of a data structure which consists of nothing else but the part. In ZIO Optics, an `Iso` is represented by the following type alias:

```
type Iso[S, A] = Optic[S, Any, A, Nothing, Nothing, A, S]
```

The simplified signature of `Optic` for `Iso` would therefore be like this:

```
case class Iso[S, A](
  getOptic: S => Either[Nothing, A],
  setOptic: A => Any => Either[Nothing, S]
)
```

Please notice the following:

- The `GetError` and `SetError` types of an `Iso` are `Nothing`, meaning that the conversion from `S` to `A` and from `A` to `S` will always succeed.
- The `GetWhole`, `SetWholeBefore`, and `SetWholeAfter` types are the same type `S`.
- The `GetPiece` and `SetPiece` types are the same type `A`.

To have a cleaner mental model of an `Iso`, we can simplify its signature even further:

```
case class Iso[S, A](
  getOptic: S => A,
  setOptic: A => S
)
```

This way, you can think of an `Iso` as an optic which converts elements of type `S` into elements of type `A` and vice versa, losslessly. And, in fact, it's called Iso because it stands for an `Isomorphism`, which is just a fancy word for Equivalence. As a curiosity, ZIO Prelude includes its own Equivalence data type.

It's also worth noticing that an `Iso` is like a combination of a `Lens` and a `Prism`, because:

- The getter in `Iso` is the same as the getter in `Lens`.
- The setter in `Iso` is the same as the setter in `Prism`.

Example with Iso

Here we have a `Color` type which just consists of one part, an rgb value:

```
final case class Color(rgb: Int)
```

We can create an `Iso` inside the Color companion object:

```
import zio.optics._

object Color {
  val iso: Iso[Color, Int] =
    Iso(
      color => Right(color.rgb),
      rgb => Right(Color(rgb))
    )
}
```

So, to create an Iso, we just need to call the Iso.apply method providing a getter (function to convert from Color to Int) and a setter (function to convert from Int to Color).

We can now use this new Iso we've just defined:

```
val color1 = Color(255)

// Get the rgb value of color1
val rgb1: Either[Nothing, Int] = Color.iso.get(color1)
// Right(255)

// Create a Color with rgb=4
val color2: Either[Nothing, Color] = Color.iso.set(4)
// Right(Color(4))
```

Finally, ZIO Optics includes a built-in Iso:

```
val identity = Iso.identity // The identity Optic
```

This `Iso` is interesting, because it represents just a pair of identity functions.

Optional

An `Optional` is a more general `Optic` than `Lens` or `Prism`, because it's used to access a piece of any ADT, where that part may or may not exist. In ZIO Optics, an `Optional` is represented by the following type alias:

```
type Optional[S, A] = Optic[S, S, A, OpticFailure, OpticFailure, A, S]
```

SollThe simplified signature of `Optic` for `Optional` would therefore now be like this:

```
case class Optional[S, A](
  getOptic: S => Either[OpticFailure, A],
  setOptic: A => S => Either[OpticFailure, S]
)
```

Please notice the following:

- The `GetError` and `SetError` types of an `Optional` are `OpticFailure` because getting or setting a piece of an ADT can fail when it does not exist.
- The `GetWhole`, `SetWholeBefore`, and `SetWholeAfter` types are the same and represent the ADT `S`.
- The `GetPiece` and `SetPiece` types are the same and represent the piece `A` we are trying to access.

Examples with Optional

Let's say we have the following `ContactInfo` ADT:

```
sealed trait ContactInfo
object ContactInfo {
  final case class Phone(number: Int)    extends ContactInfo
  final case class Email(address: String) extends ContactInfo
}
```

We can create an `Optional` for accessing the phone number, inside the `ContactInfo` companion object:

```
import zio.optics._

object ContactInfo {
  ...

  val phoneNumber: Optional[ContactInfo, Int] =
    Optional(
      {
        case Phone(number) => Right(number)
        case _             => Left(OpticFailure("Contact info does not contain a phone number!"))
      },
      newPhoneNumber =>
        contactInfo =>
          contactInfo match {
            case Phone(_) => Right(Phone(newPhoneNumber))
            case _        => Left(OpticFailure("Can't set phone number"))
          }
    )
}
```

So, to create an `Optional`, we just need to call the `Optional.apply` method providing a getter and a setter for the part we want to access (`number` inside `Phone` for this case). You can see the getter/setter can fail with an `OpticFailure` when trying to get/set a `number` from values that are not of type `Phone`.

Creating an `Optional` for accessing an email address is very similar:

```
import zio.optics._

object ContactInfo {
  ...

  val emailAddress: Optional[ContactInfo, String] =
    Optional(
      {
        case Email(address) => Right(address)
        case _              => Left(OpticFailure("Contact info does not contain an email address!"))
      },
      newEmailAddress =>
        contactInfo =>
          contactInfo match {
            case Email(_) => Right(Email(newEmailAddress))
            case _        => Left(OpticFailure("Can't set email address"))
          }
    )
}
```

We can now use these new `Optionals` we've just defined to get and set values of any `ContactInfo` object:

```
val contactInfo1 = ContactInfo.Phone(12345)
val contactInfo2 = ContactInfo.Email("test1@test.com")

// Get phone number from contactInfo1
val phoneNumber1: Either[OpticFailure, Int] = ContactInfo.phoneNumber.get(contactInfo1)
// Right(12345)

// Get phone number from contactInfo2
val phoneNumber2: Either[OpticFailure, Int] = ContactInfo.phoneNumber.get(contactInfo2)
// Left(Contact info does not contain a phone number!)

// Get email address from contactInfo1
val emailAddress1: Either[OpticFailure, String] = ContactInfo.emailAddress.get(contactInfo1)
// Left(Contact info does not contain an email address!)

// Get email address from contactInfo2
val emailAddress2: Either[OpticFailure, String] = ContactInfo.emailAddress.get(contactInfo2)
// Right(test1@test.com)



// Set new phone number to contactInfo1
val contactInfo3: Either[OpticFailure, ContactInfo] = ContactInfo.phoneNumber.set(67890)(contactInfo1)
// Right(ContactInfo.Phone(67890))

// Set new phone number to contactInfo2
val contactInfo4: Either[OpticFailure, ContactInfo] = ContactInfo.phoneNumber.set(67890)(contactInfo2)
// Left(Can't set phone number)

// Set new email address to contactInfo1
val contactInfo5: Either[OpticFailure, ContactInfo] = ContactInfo.emailAddress.set("test2@test.com")(contactInfo1)
// Left(Can't set email address)
```

```
// Set new email address to contactInfo2
val contactInfo6: Either[OpticFailure, ContactInfo] = ContactInfo.emailAddress.set("test2@test.com")(contactInfo2)
// Right(ContactInfo.Email("test2@test.com"))

// Update phone number of contactInfo1, increasing it by 1
val phoneNumber3: Either[OpticFailure, ContactInfo] = ContactInfo.phoneNumber.update(contactInfo1)(_ + 1)
// Right(ContactInfo.Phone(12346))
```

Finally, ZIO Optics includes some built-in `Optionals`, for example:

```
val head  = Optional.head        // Access the head of a List
val tail  = Optional.tail        // Access the tail of a List
val myKey = Optional.key("myKey") // Access a key of a Map
val third = Optional.at(3)       // Access an index of a ZIO Chunk
```

Traversal

A `Traversal` is an `Optic` that accesses zero or more values in a collection, such as a ZIO Chunk. In ZIO Optics, a `Traversal` is represented by the following type alias:

```
type Traversal[S, A] = Optic[S, S, Chunk[A], OpticFailure, OpticFailure, Chunk[A], S]
```

The simplified signature of `Optic` for `Traversal` would therefore now be like this:

```
case class Traversal[S, A](
  getOptic: S => Either[OpticFailure, Chunk[A]],
  setOptic: Chunk[A] => S => Either[OpticFailure, S]
)
```

Please notice the following:

- The `GetError` and `SetError` types of a Traversal are `OpticFailure` because, for example, we might be trying to get or set a value at an index that does not exist in the collection.
- The `GetWhole`, `SetWholeBefore`, and `SetWholeAfter` types are the same and represent the collection `S`.
- The `GetPiece` and `SetPiece` types are the same and represent the items `Chunk[A]` we are trying to access. This is basically what distinguishes a `Traversal` from other optics: The fact that it can access zero or more values instead of a single value.

Example with Traversal

ZIO Optics includes some constructors for `Traversals`. For instance, if we want to filter some items from a ZIO `Chunk`, we can use `Traversal.filter`. Here is an example:

```
import zio.Chunk
import zio.optics._

val filterEvenNumbers: Traversal[Chunk[Int], Int] =
  Traversal.filter(_ % 2 == 0)

val items = Chunk.fromIterable(1 to 10)

// Get all even numbers from items
val evenNumbers = filterEvenNumbers.get(items)
// Right(Chunk(2,4,6,8,10))
```

Polymorphic Optics

ZIO Optics provides more polymorphic versions of each `Optic` we have discussed previously.

For example, there is a more powerful type of `Lens`, called `ZLens`, which is represented by the following type alias:

```
type ZLens[-S, +T, +A, -B] = Optic[S, S, B, Nothing, Nothing, A, T]
```

SlThe simplified signature of `Optic` for `ZLens` would therefore now be like this:

```
case class ZLens[-S, +T, +A, -B](
  getOptic: S => Either[Nothing, A],
  setOptic: B => S => Either[Nothing, T]
)
```

And simplifying even further:

```
case class ZLens[-S, +T, +A, -B](
  getOptic: S => A,
  setOptic: B => S => T
)
```

You can see that the getter of `ZLens` is the same as the one in `Lens`. However, the setter is more powerful, because it can transform the whole of type `S` to a new whole of type `T`, by setting a value of type `B`. So, basically, a Lens is just a restricted `ZLens`, because it can't change the type of the whole data structure. The relationship between both of them is this:

```
type Lens[S, A] = ZLens[S, S, A, A]
```

Similarly, we have `ZPrism:`

```
type ZPrism[-S, +T, +A, -B] = Optic[S, Any, B, OpticFailure, Nothing, A, T]
type Prism[S, A] = ZPrism[S, S, A, A]
```

And `ZIso`:

```
type ZIso[-S, +T, +A, -B] = Optic[S, Any, B, Nothing, Nothing, A, T]
type Iso[S, A] = ZIso[S, S, A, A]
```

And `ZOptional`:

```
type ZOptional[-S, +T, +A, -B]  = Optic[S, S, B, OpticFailure, OpticFailure, A, T]
type Optional[S, A] = ZOptional[S, S, A, A]
```

And `ZTraversal`:

```
type ZTraversal[-S, +T, +A, -B] = Optic[S, S, Chunk[B], OpticFailure, OpticFailure, Chunk[A], T]
type Traversal[S, A] = ZTraversal[S, S, A, A]
```

Example of Polymorphic Optics

Let's say we have the following ADTs:

```
final case class Item[C <: Currency](description: String, price: BigDecimal, currency: C)

sealed trait Currency
object Currency {
  case object Dollar extends Currency
  case object Euro   extends Currency

  type Dollar = Dollar.type
  type Euro   = Euro.type
}
```

`Item` is a product type which models an item inside a shopping cart. What's interesting about it is that it's a polymorphic ADT, parameterized by a `C` type.

Let's say we want to create a `Lens` for accessing the `currency` inside an Item. However, we want this `Lens` not just to be able to change the value of the `currency`, but also its type, which in turn would change the type of the whole Item. For instance, if we have an `Item[Currency.Dollar]`, that means its currency is equal to `Currency.Dollar`. And, if we change the currency to `Currency.Euro`, the Item type would change to `Item[Currency.Euro]`. A simple `Lens` is not powerful enough for this case, because the setter doesn't allow us to change the type of the piece we want to set, nor the type of the whole. That means we need a `ZLens`! The way of constructing it is very similar to how we construct a `Lens`:

```
final case class Item[C <: Currency](description: String, price: BigDecimal, currency: C)
object Item {
 def currency[Old <: Currency, New <: Currency]: ZLens[Item[Old], Item[New], Old, New] =
   ZLens(
     item => Right(item.currency),
     newCurrency => item => Right(item.copy(currency = newCurrency))
   )
}
```

You can see the `ZLens` we have just defined allows us to change from an `Old` currency type to a `New` currency type.

And now, you can see the way of using a `ZLens` is exactly the same as with `Lens`:

```
val item1: Item[Currency.Dollar] = Item("Some book", 10.0, Currency.Dollar)

// Get `currency` from item1
val currency1: Either[Nothing, Currency.Dollar] = Item.currency.get(item1)
// Right(Currency.Dollar)

// Change `currency` in item1 to Euro
val item2: Either[Nothing, Item[Currency.Euro]] = Item.currency.set(Currency.Euro)(item1)
// Right(Item(Some book,10.0,Euro))
```

Composition of Optics

From the previous sections you could think that what optics essentially do, is to turn the getters and setters we know and love from OOP into values, which are type safe, principled and composable. Composability is where the real power of optics comes in. We'll now see some examples of how to use that power.

Example 1: Sequential composition of Lenses

Let's revisit the first example we looked at, in the Overview of `Optics` section:

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

In that example we wrote a `setStreetNumber` method:

```
def setStreetNumber(person: Person, newStreetNumber: Int): Person =
  person.copy(
    address = person.address.copy(
      street = person.address.street.copy(
        number = newStreetNumber
      )
    )
  )
```

Let's rewrite that method using optics. Firstly, we can define some Lenses for each case class:

```
final case class Person(fullName: String, address: Address)
object Person {
 val fullName: Lens[Person, String] =
   Lens(
     person => Right(person.fullName),
     newFullName => person => Right(person.copy(fullName = newFullName))
   )

 val address: Lens[Person, Address] =
   Lens(
     person => Right(person.address),
     newAddress => person => Right(person.copy(address = newAddress))
   )
}

final case class Address(city: String, street: Street)
object Address {
 val city: Lens[Address, String] =
```

```
  Lens(
    address => Right(address.city),
    newCity => address => Right(address.copy(city = newCity))
  )

 val street: Lens[Address, Street] =
  Lens(
    address => Right(address.street),
    newStreet => address => Right(address.copy(street = newStreet))
  )
}




final case class Street(name: String, number: Int)
object Street {
 val name: Lens[Street, String] =
  Lens(
    street => Right(street.name),
    newName => street => Right(street.copy(name = newName))
  )

 val number: Lens[Street, Int] =
  Lens(
    street => Right(street.number),
    newNumber => street => Right(street.copy(number = newNumber))
  )
```

The new version of `setStreetNumber` would be:

```
def setStreetNumber(person: Person, newStreetNumber: Int): Either[Nothing, Person] = {
  val streetNumber: Lens[Person, Int] =
    Person.address >>> Address.street >>> Street.number
  streetNumber.set(newStreetNumber)(person)
}
```

As a first step, you can see a `streetNumber Lens` has been created. The idea of this Lens is that it should allow us to access the street number inside of a `Person`. We can do that using the `>>>` operator, which allows us to do sequential composition of `Optics`, so that we can combine the Lenses we have already created for each individual case class.v

The example above is read like this:

- Create a `Lens` to access the address inside `Person`
- And then access the `street` inside `Address`
- And then access the `number` inside `Street`

The next step is just to use the `streetNumber Lens` to modify the given `person` with the given `newStreetNumber`. As you can see, this new version is more declarative and easier to understand than the original.

Example 2: Zipping two Lenses

Continuing with the previous example, what if we wanted a Lens that returned two values instead of one, for example the `fullName` and the street number where a `Person` lives? We can use the `zip` operator for that:

```
val fullNameAndStreetNumber: Lens[Person, (String, Int)] =
  Person.fullName zip streetNumber
```

Example 3: Sequential composition of Lenses and Prisms

It's important to mention that we can compose any type of optics with any other type of optics, not just `Lens` with `Lens`. This is because, as we have seen before, all of the optics in ZIO Optics are just aliases of the one and only `Optic` data type!

For instance, in the Overview of Optics section we had this other example, which is basically the same as the previous, but with fields which are of type `Option`:

```
final case class OPerson(fullName: Option[String], address: Option[OAddress])
final case class OAddress(city: Option[String], street: Option[OStreet])
final case class OStreet(name: Option[String], number: Option[Int])
```

The original implementation of `setStreetNumberOptional` was like this:

```
def setStreetNumberOptional(person: OPerson, newStreetNumber: Int): Either[String, OPerson] =
 person.address match {
   case Some(address) =>
     address.street match {
       case Some(street) =>
         Right(
           OPerson(
             fullName = person.fullName,
             address = Some(OAddress(address.city, Some(OStreet(street.name, Some(newStreetNumber)))))
           )
         )
       case None => Left("Empty street")
     }
   case None => Left("Empty address")
 }
```

And the implementation with optics would be:

```
def setStreetNumberOptional(person: OPerson, newStreetNumber: Int): Either[OpticFailure, OPerson] = {
  val streetNumber =
    OPerson.address >>> Prism.some >>>
      OAddress.street >>> Prism.some >>>
      OStreet.number >>> Prism.some
  streetNumber.set(newStreetNumber)(person)
}
```

Looks a lot better, right? The only thing we needed to do is to write a different `streetNumber Optic`. And as you can see, in this case we are composing not just `Lenses` with `Lenses`, but we also have some `Prisms` in the mix now.

The composed `Optic` would read like this:

- Create an `Optic` to access the `address` inside of `OPerson`
- And then access the `Some` case of the corresponding `Option[Address]`
- And then access the `street` inside of `Address`
- And then access the `Some` case of the corresponding `Option[Street]`
- And then access the `number` inside `Street`
- And then access the `Some` case of the corresponding `Option[Int]`

Example 4: Optional as composition of Prism and Lens

For another example of composing `Lenses` with `Prisms`, recall that in a previous section we defined an `Optional` for accessing the phone number of this ContactInfo ADT:

```
sealed trait ContactInfo
object ContactInfo {
  final case class Phone(number: Int)    extends ContactInfo
  final case class Email(address: String) extends ContactInfo
}
```

We defined the `Optional` by using the `Optional.apply` constructor. But it turns out we can also define it as a sequential composition of `Prism[ContactInfo, Phone]` and `Lens[Phone, Int]` as well. So, let's define some `Lenses` and `Prisms` first:

```
sealed trait ContactInfo
object ContactInfo {
  final case class Phone(number: Int) extends ContactInfo
  object Phone {
    val number: Lens[Phone, Int] =
      Lens(
        phone => Right(phone.number),
        newNumber => phone => Right(phone.copy(number = newNumber))
      )
  }

  final case class Email(address: String) extends ContactInfo
  object Email {
    val address: Lens[Email, String] =
      Lens(
        email => Right(email.address),
        newAddress => phone => Right(phone.copy(address = newAddress))
      )
  }

  val phone: Prism[ContactInfo, Phone] =
    Prism(
```

```
      {
        case p: Phone => Right(p)
        case _        => Left(OpticFailure("Not a Phone"))
      },
      Right(_)
    )


  val email: Prism[ContactInfo, Email] =
    Prism(
      {
        case e: Email => Right(e)
        case _        => Left(OpticFailure("Not an Email"))
      },
      Right(_)
    )
}
```

Example 5: Composing two Optics with orElse

Continuing with the previous example, what if we want to define an `Optic` that:

- Tries to access the phone number of a `ContactInfo`
- If it fails, it should try to get the email address

It turns out we can do this with the `orElse` operator, by combining the `Optionals` we have created in the previous example:

```
val phoneNumberOrEmailAddress = phoneNumber orElse emailAddress
```

Example 6: Using Optics to access deeply nested collections

For this, let's revisit the last example in the Overview of Optics section. We had a `Map` of `Customers`, indexed by customer ID:

```
val customers: Map[Long, Customer] = Map(
  1000L -> Customer("Mary Lopez", Map(1L -> Order(1, 100), 2L -> Order(2, 200))),
  2000L -> Customer("David Adams", Map(1L -> Order(3, 300), 2L -> Order(4, 400))),
  3000L -> Customer("Brian Johnson", Map(1L -> Order(5, 500), 2L -> Order(6, 600)))
)
```

The basic data structures were:

```
final case class Order(itemId: Long, quantity: Long)
final case class Customer(name: String, orders: Map[Long, Order])
```

We implemented a `setQuantity` method to modify the `quantity` of an `Order`, given the customer ID and order ID. Our implementation had lots of boilerplate, so now let's use ZIO Optics instead. First, let's create some Lenses for `Order` and `Customer`:

```
final case class Order(itemId: Long, quantity: Long)
object Order {
  val quantity: Lens[Order, Long] =
    Lens(
      order => Right(order.quantity),
      newQuantity => order => Right(order.copy(quantity = newQuantity))
    )
}

final case class Customer(name: String, orders: Map[Long, Order])
object Customer {
  val orders: Lens[Customer, Map[Long, Order]] =
    Lens(
      customer => Right(customer.orders),
      newOrders => customer => Right(customer.copy(orders = newOrders))
    )
}
```

We can now rewrite the `setQuantity` method:

```
def setQuantity(
 customers: Map[Long, Customer],
 customerId: Long,
 orderId: Long,
 newQuantity: Long
): Either[OpticFailure, Map[Long, Customer]] = {
  val quantityOptic =
    Optic.key[Long, Customer](customerId) >>> Customer.orders >>> Optic.key(orderId) >>> Order.quantity
  quantityOptic.set(newQuantity)(customers)
}
```

This new solution is a lot more elegant than the original, and again we are just using optics composition. The composed `quantityOptic` is read like this:

- Create an `Optic` to access the `customerId` key inside a `Map[Long, Customer]`
- And then access the `orders` inside the corresponding `Customer`
- And then access the `orderId` key inside the corresponding `Map[Long, Order]`
- And then access the `quantity` inside `Order`

Example 7: Using the foreach operator of Traversal

It turns out that, when we have a `Traversal`, we can call the `Traversal#foreach` operator on it, which expects another `Optic` to be applied to each item accessed by the aforementioned `Traversal`.

For example, let's say we have a `Chunk[Person]`, and we want to get the names of every Person who lives in New York. We already have the Person type from previous examples, and we've also already defined `Lenses` for it, so here's how we would solve this problem by composing `Optics`:

```
def getPeopleNamesFromNewYork(
  people: Chunk[Person]
): Either[OpticFailure, Chunk[String]] = {
  val newYorkNames =
    Traversal.filter[Person](_.address.city == "New York").foreach(Person.fullName)
  newYorkNames.get(people)
}
```

You can see we have defined a `newYorkNames Optic`, which uses the `Traversal#foreach` operator to compose a Traversal.filter optic with a Lens we already have: `Person.fullName.` After the Optic is defined, we can just use it to get the desired items from the given `people`.

So, if we have:

```
val people = Chunk(
  Person("John Adams", Address("New York", Street("Some street", 100))),
  Person("Juanita Perez", Address("Los Angeles", Street("Another street", 500))),
  Person("Lucy Smith", Address("New York", Street("Some street", 200))),
  Person("Andrew Johns", Address("San Diego", Street("The street", 600)))
)
```

Then, the result of calling `getPeopleNamesFromNewYork` would be:

```
getPeopleNamesFromNewYork(people)
// Right(Chunk(John Adams,Lucy Smith))
```

Special support for ZRef

ZIO Optics provides special support for ZRef, by adding some extension methods to it which use optics under the hood. There are several extension methods according to the contents inside the `ZRef`. For example, if it contains a `Chunk`:

```
val chunk: Ref[Chunk[Int]] = ???

chunk.at(10) // Access the specified index of `chunk`
chunk.filter(_ % 2 == 0) // Filter elements of a `chunk`
chunk.slice(2, 5) // Access a slice of `chunk`
```

If it contains a `List`:

```
val list: Ref[List[Int]] = ???

list.cons // Access the `::` case of `list`
list.head // Access the head of `list`
list.tail // Access the tail of `list`
```

If it contains a `Tuple2`:

```
val tuple: Ref[(Int, String)] = ???

tuple.first  // Access the first element of `tuple`
tuple.second // Access the second element of `tuple`
```

If it contains a `Map`:

```
val map: Ref[Map[String, Int]] = ???

map.key("someKey") // Access the given key of `map`
```

If it contains an `Either`:

```
val either: Ref[Either[String, Int]] = ???

either.left  // Access the Left case of `either`
either.right // Access the Right case of `either`
```

And, it it contains an `Option`:

```
val option: Ref[Option[Int]] = ???

option.none // Access the None case of `option`
option.some // Access the Some case of `option`
```

Let's see a concrete example of how useful this is. Let's say we have a `Ref` containing a `Map` whose keys are movie titles and whose values are movie descriptions:

```
type Title       = String
type Description = String

val movieDescriptionsByTitleRef: Ref[Map[Title, Description]] = ???
```

We want to write a method to update the description of the given movie. Without ZIO Optics, it would look like this:

```
def updateMovieDescription(title: Title, newDescription: Description): Task[Unit] =
  movieDescriptionsByTitleRef.modify { movieDescriptionsByTitle =>
    movieDescriptionsByTitle.get(title) match {
      case Some(_) =>
        (ZIO.unit, movieDescriptionsByTitle + (title -> newDescription))
      case None =>
        (
          ZIO.fail(new NoSuchElementException(s"Movie with title $title not found")),
          movieDescriptionsByTitle
        )
    }
  }.flatten
```

But with ZIO Optics, we have a cleaner and more declarative implementation:

```
import zio.optics._

def updateMovieDescription2(title: String, newDescription: String): Task[Unit] =
  movieDescriptionsByTitleRef.key(title).update(_ => newDescription)
```

Effectful and Transactional Optics

So far we've explored Pure Optics, but as was previously mentioned, ZIO Optics also allows us to work with Effectful and Transactional Optics.

Effectful Optics work exactly the same way as Pure Optics, but the `OpticResult` returned by the getter and setter is a ZIO effect, so we have:

```
case class Optic[
  -GetWhole,
  -SetWholeBefore,
  -SetPiece,
  +GetError,
  +SetError,
  +GetPiece,
  +SetWholeAfter
](
  getOptic: GetWhole => IO[(GetError, SetWholeAfter), GetPiece],
  setOptic: SetPiece => SetWholeBefore => IO[(SetError, SetWholeAfter), SetWholeAfter]
)
```

To work with Effectful `Optics`, just include the following import:

```
import zio.optics.opticsm._
```

Transactional Optics also work exactly the same as Pure Optics, but the `OpticResult` returned by the getter and setter is an STM effect, so we have:

```
case class Optic[
  -GetWhole,
  -SetWholeBefore,
  -SetPiece,
  +GetError,
  +SetError,
  +GetPiece,
  +SetWholeAfter
](
  getOptic: GetWhole => STM[(GetError, SetWholeAfter), GetPiece],
  setOptic: SetPiece => SetWholeBefore => STM[(SetError, SetWholeAfter), SetWholeAfter]
)
```

To work with Transactional Optics, just include the following import:

```
import zio.optics.toptics._
```

So, for example, let's say we have a nested TMap (which is a data structure from ZIO STM) like this:

```
import zio.stm._

val tmap: USTM[TMap[String, Either[String, Int]]] =
  TMap.make(
    ("test1", Left("hello")),
    ("test2", Right(1))
  )
```

And we want to write a method which, given a `TMap` and `key`, increments the corresponding value by 1, in the case the value is an integer. Without ZIO Optics, we would have something like this:

```
def updateMap(
  tmap: TMap[String, Either[String, Int]],
  key: String
): STM[String, TMap[String, Either[String, Int]]] =
  for {
    optionEither <- tmap.get(key)
    either       <- STM.fromOption(optionEither) <> STM.fail(s"tmap does not contain key $key")
    _ <- either match {
          case Left(_)    => STM.fail("Not an Int")
          case Right(int) => tmap.put(key, Right(int + 1))
        }
  } yield tmap
```

But, with ZIO Optics we have:

```
import zio.optics.toptics._

def updateMap2(
 tmap: TMap[String, Either[String, Int]],
 key: String
): STM[OpticFailure, TMap[String, Either[String, Int]]] = {
  val optic: Optional[TMap[String, Either[String, Int]], Int] =
    TOptics.key(key) >>> Optic.right
  optic.update(tmap)(_ + 1)
}
```

Looks a lot better! And as you can see, the way to work with Transactional Optics is pretty much the same as when we work with Pure Optics. The only new thing here is we have `TOptics.key`, which is a specialized `Prism` that works with `TMap`.

Coming soon to ZIO Optics

So far we've seen that ZIO Optics already provides lots of nice tools that developers can use to improve their productivity. However, ZIO Optics is still in the development stage, so there is even more great stuff coming in the future:

- Automatic generation of optics for any ADT: In the previous examples you can see that, if you want the power of optics, you will have to write basic optics (such as `Lenses` or `Prisms`) for all of the fields of your product/sum types. That means the more fields and ADTs you have, the more optics you'll need to write. The good news is that, in future releases, ZIO Optics will automatically derive these for you, so you won't have to write them yourself!

- Dot syntax: Previously we have seen how ZIO Optics adds extension methods to the `ZRef` data type from ZIO. This allows us to work with a nice dot syntax, like this:

```
ref.key("key").right.at(0).update(_ + 1)
```

In future releases, it should be possible to have a similar dot syntax for ordinary values, as shown in the [ZIO Optics documentation]:

```
val map: Map[String, Either[String, Chunk[Int]]] = ???

val updated: Either[OpticFailure, Map[String, Either[String, Chunk[Int]]]] =
  map.optic.key("key").right.at(0).update(_ + 1)
```

Moreover, when automatic derivation of optics is introduced, this syntax will be supported for user defined data structures as well.

- Constructors and operators for other types of optics that can be derived from the basic `Optic` type: `Getter, Setter, Fold.` If you want to get a basic idea about these, you can take a look at the [ZIO Optics documentation].

- More constructors and operators for `Lens, Prism, Iso, Optional` and `Traversal.`

Summary

In this document, you have seen the importance of optics, a very powerful tool in Functional Programming. It allows us to work with deeply nested immutable data structures in a type-safe, principled and composable way. This will help us a lot when it comes to reducing boilerplate code in our applications. We've also seen that ZIO Optics provides a very powerful and flexible Optic data type, from which all the different kinds of optics are derived: `Lens`, `Prism`, `Iso`, `Optional` and `Traversal`, including polymorphic versions.

But not only all that. We've also seen how, thanks to its modular design, ZIO Optics provides not just Pure Optics but also Effectful and Transactional Optics, which enable a whole new space of possibilities by having optics that can perform ZIO effects or participate in ZIO STM transactions. Moreover, ZIO Optics is completely extensible, so it can be extended to interact with other effect systems such as Cats-Effect or Monix.

I hope the concepts and examples presented in this document will be useful to you, so you can start using ZIO Optics in your own applications!

Finally, take a look at this GitHub repository which contains all of the code samples shown above. You could also see this great Zymposium presentation by Adam Fraser and Kit Langton where they presented ZIO Optics for the very first time.

References

- **GitHub repository for this document**
- **ZIO Optics GitHub repository**
- **ZIO Optics official documentation**
- **ZIO Optics Zymposium presentation** by Adam Fraser and Kit Langton


scalac