Asymmetric Lenses in Scala

Tony Morris
tmorris@tmorris.net

Abstract

Lenses are bidirectional transformations between pairs of connected structures. Asymmetric lenses —where one of those two connected structures is taken to be primary —have been extensively studied. Lenses were first proposed to solve the view-update problem of tree-like data structures by Foster et. al[4] and have also been applied to the construction of a relational database query language[2]. Other work has altered the primitive structure of lenses to achieve different results[5].

We begin by exploring the existing utilities available for bidirectional transformations in the Scala programming language. We discuss the associated problems with these utilities using examples and propose a solution using asymmetric lenses. We present the canonical representation of an asymmetric lens and an alternative representation which is particularly suited to the specific properties of the Scala programming language and which has desirable properties for users.

We explore a small set of interesting user libraries that arise as a consequence of integrating asymmetric lenses into Scala, chosen for the purpose of demonstrating utility. We will also take a brief look at partial lenses —applicable to sum types, which Scala directly supports. Finally, we briefly examine existing requirements, solutions and proposals for integrating lenses into the Scala programming language.

Categories and Subject Descriptors D [3]: 2

General Terms Functional Programming, Composition, Languages, Scala

Keywords Lens, Comonad, Monad, Coalgebra, CoState, Store, State, Category, Monoid

1. Introduction

Scala includes a language feature whereby a **`case class`**[1] produces a compiler-generated accessor for each of its fields and similarly, copy method syntax for updating fields of a case class. This language feature is similar in function to SML or Haskell record selectors.

[1] also referred to as *records* in a programming language-agnostic nomenclature

Scala Days '12 April 17 —18, 2012, London, UK.

For example, given the following definition:

Listing 1: Person.scala

```
case class Address(
  street: String
, state:  String
)

case class Person(
  age:     Int
, address: Address
)
```

On a given instance of `Person`, call it `p`, we may access its `age` field by using its generated accessor method of the same name as the field:

```
val a: Int = p.age
```

Also, we may update its `age` field to the value `x`, returning a new `Person` object, by using the generated copy method syntax:

```
val updatedPerson: Person = p.copy(age = x)
```

While these abilities are available to the Scala programmer with no intervention after the declaration of the **`case class`**, the generated accessor/updater functions are not *first-class* values. This inhibits the programmer from implementing general libraries and in practice, results in significant boilerplate code.

2. The Problem

A trivial use-case to demonstrate the boilerplate problem is that of defining a *modify* function for a given field of a record. The modify function is used to pass a function and a record and it returns a new record having updated the field with that function. The type signature of the modify function takes on the form:

Listing 2: The signature of a `modify` function

```
def modify: (Field => Field) => Rec => Rec
```

For example, taking the running example of the `age` field of the `Person` record[2]:

Listing 3: The `modifyAge` function

```
def modifyAge
  (f: Int => Int)
  (p: Person)
  : Person =
    p.copy(age = f(p.age))
```

[2] This definition could also be written as an instance method of the `Person` class.

A subsequent definition of a modify function for the `address` field of the `Person` class would require similar boilerplate code:

Listing 4: The `modifyAddress` function

```
def modifyAddress
  (f: Address => Address)
  (p: Person)
  : Person =
    p.copy(address = f(p.address))
```

Indeed, there are potentially as many modify functions as there are record fields, each requiring programmer intervention to implement.

However, the problem is even more pronounced. Let us suppose we wish to access the `street` field of the `address` field of a `Person`.

Listing 5: The `getPersonStreet` function

```
def getPersonStreet(p: Person): String =
  p.address.street
```

Similarly, we may wish to update the `street` field of the `address` field of a `Person`.

Listing 6: The `setPersonStreet` function

```
def setPersonStreet
  (p: Person)
  (s: String)
  : String =
    p.copy(address =
      p.address.copy(street = s))
```

There is also another potential modify function for a Person's (indirect) street field. This particular undesirable consequence is not experienced when modelling the program design to use mutable fields, however, if we are to strive for the benefits of immutable records, we need to find a way to alleviate this issue of laborious boilerplate code.

In summary, the problem grows proportional to $M \times N$ where:

- M = The number of potential record fields.
- N = The number of potential library functions on record fields in general.[3]

A cursory examination of a record's generated accessor and copy method gives us functions with the types:

- `get: Rec => Field`
- `set: (Rec, Field)=> Rec`

For example, the `age` accessor and copy method have the types:

- `Person => Int`
- `(Person, Int)=> Person`

We will now take this observation and explore its utility in detail.

3. Asymmetric Lenses

3.1 Canonical Representation

An asymmetric lens is given by a binary type constructor defined over the record type (R) —the primary component in the bidirectional relationship —and the field type (F). The lens structure is the product of the accessor and updater function:

Listing 7: Lens.scala

```
case class Lens[R, F](
  get: R => F
, set: (R, F) => R
)
```

We would then denote the lens of a `Person` to its `age` with a value of the `Lens` type:

Listing 8: ageLens

```
val ageL: Lens[Person, Int] =
  Lens(
    get = _.age
  , set = (p, a) => p.copy(age = a)
  )
```

We are now able to derive a `modify` function that operates on any lens.

Listing 9: The `modify` function derivable from any `Lens`

```
def modify[R, F]
  (l: Lens[R, F])
  (f: F => F)
  : R => R =
    r => l set (r, f(l get r))
```

3.2 Lens Laws

Lens implementations are expected to satisfy three laws (invariants). Adherence to these laws provides guarantees about the behaviour of higher-level libraries on lenses. The Scala typechecker will not enforce these laws so adherence is delegated to the programmer's responsibility.

1. If you set a field, then get that field, you have the originally set field value:

   ```
   ∀ r f. lens.get(lens.set(r, f)) == f
   ```

2. If you get a field, then set that field, you have the same record value:

   ```
   ∀ r. lens.set(r, lens get r) == r
   ```

3. If you set a field twice with values `f2` then `f1`, this is the same as setting it once with value `f1`. The first set operation of `f2` is not observable.

   ```
   ∀ r f1 f2.
     lens.set(lens.set(r, f2), f1) ==
       lens.set(r, f1)
   ```

[3] We will go further into these library functions later in section 5.

We can express these laws in Scala. All the following functions should return **true**, regardless of the value of their arguments:

Listing 10: Lens laws expressed in Scala

```
object LensLaws {
  def law1[R, F]
    (r: R, f: F)
    (lens: Lens[R, F]) =
      lens.get(lens.set(r, f)) == f

  def law2[R, F](r: R)(lens: Lens[R, F]) =
    lens.set(r, lens get r) == r

  def law3[R, F]
    (r: R, f1: F, f2: F)
    (lens: Lens[R, F]) =
      lens.set(lens.set(r, f2), f1) ==
        lens.set(r, f1)
}
```

3.3 Lenses are Categories

Perhaps the first interesting observation about a `Lens` is that it forms a category[1]. We take a category by the following interface:

Listing 11: The Category trait

```
trait Category[~>[_, _]] {
  def compose[A, B, C]
    (f: B ~> C)
    (g: A ~> B)
    : A ~> C
  def id[A]: A ~> A
}
```

We implement the `Category` trait for `Lens` by the following implementation:

Listing 12: The Category instance for Lens

```
val lensCat =
  new Category[Lens] {
    def compose[A, B, C]
    (f: Lens[B, C])
    (g: Lens[A, B]) =
      Lens(
        get =
          f.get compose g.get
      , set = (a, c) =>
          g set(a, f set (g get a, c))
      )
    def id[A]: Lens[A, A] =
      Lens(
        get = identity
      , set = (a, _) => a
      )
  }
```

3.4 Lenses Compose

We can see immediately that since a `Lens` composes in the same way that `scala.Function1` composes[4], we can produce a new `Lens` for embedded data structures by composing each lens for each level of embedding. We add a method `compose` representing composition under the lens category:

Listing 13: Lens with `compose` method

```
case class Lens[R, F](
  get: R => F
, set: (R, F) => R
) {
  import Lens._

  def compose[Q]
    (g: Lens[Q, R])
    : Lens[Q, F] =
      Lens(
        get =
          get compose g.get
      , set = (q, f) =>
          g set (q, set(g get q, f))
      )
}
```

3.5 Exploiting Lens Composition

An example of exploiting the composition of lenses is to produce the lens for the street of a person's address[5]. We may first take the address lens, which has the type `Lens[Person, Address]`. We then take the street lens, which has the type `Lens[Address, String]` and we can produce a lens with the type `Lens[Person, String]` by exploiting composition:

Listing 14: Lenses under composition

```
def addressL: Lens[Person, Address] = ...
def streetL: Lens[Address, String] = ...
val personStreetL: Lens[Person, String] =
  streetL compose addressL
```

Using the `personStreetL` lens, we may access or set the (indirect) street property of a `Person` instance.

- ```
  val str: String =
    personStreetL get person

  val newP: Person =
    personStreetL set (person, "Bob␣St")
  ```

Other interesting functions arise as a consequence of the lens category. For example, since all categories are monoids, we may reduce a list of lenses:

Listing 15: Fold a list of lenses

```
def foldLens[A]
  (x: List[Lens[A, A]])
  : Lens[A, A] =
    x.foldRight(lensCat.id)(_ compose _)
```

4. Representing an Asymmetric Lens in Scala

So far, we have denoted a lens by the product of the `get` and `set` functions.

There exists an alternative representation of asymmetric lenses which exploits a fusion of the target object onto the pair

[4] See the `scala.Function1#compose` method for its similarities —it too, forms a category.

[5] Recall that a person has an address and an address has a street.
```

of the `get` and `set` functions —assuming a high likelihood that both `get` and `set` will be called on a given target object. This representation was first described by Russell O'Connor[9]. Although this alternative representation is *isomorphic* to our canonical representation and one can replace the other without affecting client code, it provides a couple of useful benefits:

- obviates the existence of particular library functions on structures used in the representation. The `CoState` data structure —used for representation —provides many library utilities, including a comonadic interface.
- suited to the Scala programming environment by providing a performance enhancement as a consequence of the fusion on the target object, increasing the viability of lenses on the JVM. In particular, since the application to the target object occurs once, we have a more efficient `modify` operation by having direct access to `get` and `set` closed over the target object. Many higher-level lens libraries utilise the `modify` operation and so would acquire this benefit.

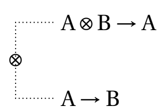

Figure 1: Canonical representation of an asymmetric lens.

This described alternative lens representation is first arrived at by currying the `set` function.

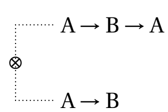

Figure 2: Intermediate step in an alternative representation of an asymmetric lens. The `set` operation is curried.

We can see in this intermediate step that there is a common argument —the target object —to both function pairs. Let us now fuse the pair of functions on this argument.

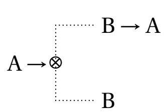

Figure 3: Alternative representation of an asymmetric lens fused on the target object.

4.1 Fusing the Target of a Lens

This alternative representation provides the user with an interface where the target object is passed and a pair denoting the `get` and `set` functions is returned, closed over the target object. We will call this pair of functions, `CoState`.

Listing 16: An asymmetric lens fused on the target object

```
case class CoState[F, R](
  get: F
, set: F => R
)

case class FLens[R, F](
  apply: R => CoState[F, R]
)
```

4.2 A Lens Fused on its Target is Isomorphic

To demonstrate that this alternative representation has not altered the algebraic structure of the lens, the isomorphism is given by the following bijection:

Listing 17: The isomorphism of the two lens representations

```
object LensIso {
  def ==>[R, F]
    (x: Lens[R, F])
    : FLens[R, F] =
      FLens(r =>
        CoState(x get r, x.set(r, _)))
  def <==[R, F]
    (x: FLens[R, F])
    : Lens[R, F] =
      Lens(
        x apply _ get
      , x apply _ set _
      )
}
```

Further, by exploring the algebraic representation of each structure, we can observe an equivalence by specification testing.

Under the usual algebraic correspondence, taking => to denote exponentiation and pairing to denote multiplication, the canonical representation of a lens gives us:

$$F^R \times R^{F \times R}$$

Similarly, for our alternative representation, we have:

$$\left(F \times R^F\right)^R$$

We can assert this using ScalaCheck[6]:

Listing 18: Verifying the isomorphism with ScalaCheck

```
forAll((r: Double, f: Double) =>
  (r >= 0 && f >= 0) ==> {
    val p = math.pow(_: Double, _: Double)
    val canonical = p(f, r) * p(r, f * r)
    val alternative = p(f * p(r, f), r)
    canonical == alternative
  }
).check

+ OK, passed 100 tests.
```

We will revert to using `Lens` from hereon, keeping in mind that `FLens` is substitutable by the bijection.

[6] `http://code.google.com/p/scalacheck/`

4.3 CoState is a Comonad

`CoState` is so named because it is the *categorical dual* of the relatively familiar `State` data structure[6]. Just as `State` is a (particularly useful) monad and so gives rise to `flatMap` and `map` methods, so too `CoState` is a comonad[9], giving rise to `coFlatMap` [7] and `map` methods.

Listing 19: State monad and CoState comonad

```
case class State[S, A](run: S => (A, S)) {
  def map[B](f: A => B): State[S, B] =
    State(s => {
      val (a, t) = run(s)
      (f(a), t)
    })

  def flatMap[B]
    (f: A => State[S, B])
    : State[S, B] =
      State(s => {
        val (a, t) = run(s)
        f(a) run t
      })
}

case class CoState[F, R](
  get: F
, set: F => R
) {
  def map[S](f: R => S): CoState[F, S] =
    CoState(
      get
    , f compose set
    )

  def coFlatMap[S]
    (f: CoState[F, R] => S)
    : CoState[F, S] =
      CoState(
        get
      , k => f(CoState(k, set))
      )
}
```

As a consequence of this comonad, we have access to libraries that naturally arise. For example, we are able to run the co-flatten[8] operation —the dual of the more familiar `flatten` operation in Scala —on a `CoState` instance:

Listing 20: Comonadic duplication of the get and set pair of `CoState`

```
def coFlatten[F, R]
  (s: CoState[F, R])
  : CoState[F, CoState[F, R]] =
    s coFlatMap identity
```

[7] so named to keep with the Scala convention of `flatMap` but the type signature has "the arrows reversed"

[8] `coFlatten: F[A] => F[F[A]]` in contrast to `flatten: F[F[A]] => F[A]`

5. Lens Libraries

So far, we have visited a few useful libraries on lenses:

- The `modify` function, available on any `Lens`.
- The `compose` function —a consequence of `Lens` being an instance of a category.
- The `fold` function —a consequence of all categories being an instance of a monoid.
- The `Comonad` instance of the `CoState` data structure.
- The `coFlatten` function —a consequence of `CoState` being an instance of a comonad.

However, this list is relatively brief as the library that can be derived from the uses of lenses is very rich. Let us explore some of those libraries.

5.1 Lens Product Morphism

Let us take two disjoint lenses[9] and combine them to produce a new lens by pairing their input and output $(R \rightsquigarrow F) \rightarrow (S \rightsquigarrow G) \rightarrow (R \otimes S \rightsquigarrow F \otimes G)$. This is useful for building up lenses into products of lenses. The type of this function is

```
Lens[R, F]
=> Lens[S, G]
=> Lens[(R, S), (F ,G)]
```

and is implemented as an instance method on `Lens[R, F]`.

Listing 21: Taking the product of two lenses to produce a new lens

```
case class Lens[R, F](
  get: R => F
, set: (R, F) => R
) {
  def ***[S, G]
    (y: Lens[S, G])
    : Lens[(R, S), (F, G)] =
      Lens(
        rs => (get(rs._1), y get rs._2)
      , (rs, fg) =>
          (
            set(rs._1, fg._1)
          , y set (rs._2, fg._2)
          )
      )
}
```

[9] That is, unrelated lenses with completely different types.

5.2 Lens Choice Morphism

We may take two lenses that view the same field type and join them into a choice on the record $(R \rightsquigarrow F) \rightarrow (S \rightsquigarrow F) \rightarrow (R \oplus S \rightsquigarrow F)$.

The type of this function is

```
Lens[R, F]
=> Lens[S, F]
=> Lens[Either[R, S], F]
```

and is implemented as an instance method on `Lens[R, F]`.

Listing 22: Taking two lenses to choose on the record type to produce a new lens viewing the same field type

```
case class Lens[R, F](
  get: R => F
, set: (R, F) => R
) {
  def |||[S, G]
    (y: Lens[S, F])
    : Lens[Either[R, S], F] =
      Lens({
        case Left(r)  => get(r)
        case Right(s) => y get s
      }
      , {
        case (Left(r), f)  =>
          Left(set(r, f))
        case (Right(s), f) =>
          Right(y.set(s, f))
      })
}
```

5.3 Lens Codiagonal Morphism

A lens gives rise to a codiagonal morphism $A \oplus A \rightsquigarrow A$. This may be thought of as taking either A or A and stripping the choice of A from the `Either` value. The type of this value is `Lens[Either [A, A], A]`.

Listing 23: Lens is codiagonal

```
def codiag[A]: Lens[Either[A, A], A] = {
  val id = Lens[A, A](identity, (_, a) => a)
  id ||| id
}
```

5.4 Standard Lenses

Some lenses are relatively mundane, operating on standard library data types. For example, the lens to get/set the first or second side of a pair:

Listing 24: Lens on Pairs

```
def first[A, B]: Lens[(A, B), A] =
  Lens(_._1, (ab, a) => (a, ab._2))
def second[A, B]: Lens[(A, B), B] =
  Lens(_._2, (ab, b) => (ab._1, b))
```

Others operate on collections, such as `Map` or `Set`:

Listing 25: Map and Set lenses

```
def mapL[K, V](k: K)
    : Lens[Map[K, V], Option[V]] =
  Lens(
    _ get k
  , (m, v) => v match {
      case None => m - k
      case Some(w) => m + ((k, w))
    }
  )

def setL[K](k: K)
    : Lens[Set[K], Boolean] =
  Lens(
    _ contains k
  , (s, p) => if(p) s + k else s - k
  )
```

Several other standard lenses operate on:

- queues —enqueue, dequeue and length operations.
- arrays —setting and retrieving a value at an index.
- numeric values —performing numeric operations.

5.5 Infix Type Alias

Since `Lens` is a binary type constructor and Scala has support for infix types[10], a lens library may benefit from an infix type alias. For example, the alias `@>` may be used to denote a lens and its asymmetry.

```
type @>[A, B] = Lens[A, B]
```

This type alias is purely aesthetic and offers no computational power, however, some expressions may benefit from this aesthetic improvement. For example, consider the type of lens composition:

```
(B @> C) => (A @> B) => (A @> C)
```

This type signature is similar to the type of regular function composition:

```
(B => C) => (A => B) => (A => C)
```

Indeed, the signature is the type of composition under a category, where that category is `@>` denoting the lens category in this instance and the `=>` category in the case of function composition.

5.6 Lens Produces State

A lens gives rise to a trivial `State` instance by

```
l => State(s => (l get s, s))
```

Since `State` forms a monad, this gives rise to other useful programming constructs, such as emulating imperative programming (5.7).

5.7 Emulating Imperative Programming

Lenses, combined with state and its monad provides a facility to emulate imperative programming without losing the program properties that allow us to reason equationally about our program i.e. we maintain referential transparency. Following is the set-up of a use-case by introducing a small subset of the typical lens/state libraries.

Let us first add two methods to lenses:

1. `+=` which operates on any numeric field of a record by adding a given value and boxing that computation in `State`.
2. `:=` which updates the field of a record and boxing that computation in `State`.

[10] Scala Language Specification 3.2.8 Infix Types

Listing 26: Lenses with methods for imperative programming

```
case class Lens[R, F](
  get: R => F
, set: (R, F) => R
) {
  def +=(n: F)
    (implicit m: Numeric[F])
    : State[R, F] =
      State(r => {
        val w = m.plus(get(r), n)
        (w, set(r, w))
      })

  def :=(f: => F): State[R, F] =
    State(r => (f, set(r, f)))
}
```

Next, we examine some combinators on `State`, which will be useful for our use-case:

1. `get` which gets the current state value into the non-state value of `State`. That is, it duplicates the state value with `s => (s, s)`.
2. `eval` which executes the `State` and discards the resulting state value, leaving only the non-state value.
3. `st` which implicitly lifts any lens into a `State` value —the trivial one mentioned earlier (5.6).

Listing 27: State combinator functions

```
def get[S]: State[S, S] =
  State(s => (s, s))
def eval[S, A](t: State[S, A])(s: S): A =
  t.run(s)._1
implicit def st[R, F]
  (l: Lens[R, F])
  : State[R, F] =
    State(s => (l get s, s))
```

Let us now define the record on which we will deploy our use-case:

```
case class Employee(
  name: String
, salary: Int
, age: Int
)
```

Now, we must hand-roll our lenses for each field of the employee record. We will look at techniques for integrating lenses into Scala so to avoid the need for this boiler-plate in section 7.

```
val name =
  Lens(
    _.name
  , (e, n) => e copy (name = n)
  )
val salary =
  Lens(
    _.salary
  , (e, s) => e copy (salary = s)
  )
val age =
  Lens(
    _.age
  , (e, a) => e copy (age = a)
  )
```

Finally, we demonstrate the use-case by:

- updating an employee's salary by `100`
- updating an employee's name by appending a surname `"␣Jones"`

We do this by keeping these modifications inside a state value, which is constructed by composing smaller state values with its monad. The final modification is kept in a value, which may then be applied to any `Employee` instance.

Listing 28: Imperative programming with lenses and state

```
val modification =
  for {
    _ <- salary += 100
    n <- name
    _ <- name := n + "␣Jones"
    e <- get
  } yield e

val bill = Employee("Bill", 1100, 33)
val updatedBill = eval(modification)(bill)
println(updatedBill)
```

The above program prints:

```
> Employee("Bill␣Jones", 1200, 33)
```

We note that the employee has had the modifications performed without in-place updates and by combining small, reusable program parts to produce the use-case. With sufficient library support, and higher-level abstractions, non-trivial use-cases can be easily and robustly handled.

This example, and all associated code mentioned so far, is available to download and run from Github [8].

6. Partial Lenses

Just as regular lenses correspond to fields of record types, *partial lenses* correspond to constructors of sum types. Partial lenses are represented by a function accepting the target object and *optionally* returning a value with the type `CoState[C, T]` where `C` denotes the type of the constructor's arguments and `T` denotes the type that the constructor produces.

For example, the `Option` partial lens defined for the `Some` constructor produces (an optional) value of the type `CoState[A, Option[A]]` since the `Some` constructor accepts a value of the type `A` to construct a value of the type `Option[A]`. This partial lens value is implemented further on in listing 31.

Listing 29: Partial Lens data structure

```
case class PartialLens[T, C](
  apply: T => Option[CoState[C, T]]
)
```

Just as a mundane example of the use of regular lenses is for the most basic product type, `Tuple2` and its `_1` and `_2` fields, a similar example of the use of partial lenses is for the most basic sum type, `Either` and its `Left` and `Right` constructors.

Listing 30: Partial Lens for constructors of `Either`

```
def leftPartialLens[A, B]
    : PartialLens[Either[A, B], A] =
  PartialLens {
    case Left(a)  =>
      Some(CoState(a, Left(_)))
    case Right(_) =>
      None
  }

def rightPartialLens[A, B]
    : PartialLens[Either[A, B], B] =
  PartialLens {
    case Right(b) =>
      Some(CoState(b, Right(_)))
    case Left(_)  =>
      None
  }
```

Informally, we may encounter problem questions in the field such as:

- “How do I update the left value of my Either?”
- “How do I update the head of my list?”
- “How do I modify the first element of my JSON Array?”

Of course, there is not necessarily a left value of an `Either` (it may be `Right`) or there may not be a head of a list (it may be empty), however, partial lenses allow us to program in such a way as if those values did exist, with the partial lens itself taking care of the possibility that it might not. Indeed, partial lenses provide an elegant answer to the aforementioned questions.

Following are more interesting partial lens values defined on data types that are integral to the Scala standard library —some oriented toward the above questions:

Listing 31: Partial Lens for list head, `Option Some`, JSON array

```
// the head of a list
def listHeadPartialLens[A]
    : PartialLens[List[A], A] =
  PartialLens {
    case Nil => None
    case h :: t => Some(CoState(h, _ :: t))
  }

// the Some value of an Option
def somePartialLens[A]
    : PartialLens[Option[A], A] =
  PartialLens(_ map (z =>
    CoState(z, Some(_))))

// the array of a JSON value
// (util.parsing.json)
def scalaJSONArrayPartialLens[A]
    : PartialLens[JSONType, List[Any]] =
  PartialLens {
    case JSONArray(a) =>
      Some(CoState(a, JSONArray(_)))
    case _            =>
      None
  }
```

Partial lenses exhibit many of the same properties of regular lenses:

- Partial lenses form a category, with the ability to compose and an identity value. $(P \rightsquigarrow Q) \to (Q \rightsquigarrow R) \to (P \rightsquigarrow R)$ and $(P \rightsquigarrow P)$
- Partial lenses can alternate on a choice. $(R \rightsquigarrow F) \to (S \rightsquigarrow F) \to (R \oplus S \rightsquigarrow F)$
- Partial lenses combine on the product of disjoint values. $(R \rightsquigarrow F) \to (S \rightsquigarrow G) \to (R \otimes S \rightsquigarrow F \otimes G)$
- Partial lenses give rise to a codiagonal morphism. $A \oplus A \rightsquigarrow A$
- Partial lenses give rise to a `State` value, useful for monadic programming, for which Scala has syntactic support.
- Partial lenses exhibit a `modify` operation, for running modifications on constructor values.

Implementations for these operations follow.

Listing 32: Partial Lens with common lens operations

```
case class PartialLens[T, C](
  apply: T => Option[CoState[C, T]]
) {
  // Partial lens category composition
  def compose[D](f: PartialLens[C, D])
      : PartialLens[T, D] =
    PartialLens(t => for {
      c <- this apply t
      d <- f apply c.get
    } yield CoState(
      d.get
    , x => c set (d set x)
    ))
  // Alternate on a choice
  def |||[U](f: PartialLens[U, C])
      : PartialLens[Either[T, U], C] =
    PartialLens {
      case Left(a) =>
        apply(a) map (x => CoState(
          x.get
        , w => Left(x set w)
        ))
      case Right(b) =>
        f apply b map (y => CoState(
          y.get
        , w => Right(y set w)
        ))
    }
  // Product of disjoint lenses
  def ***[U, D](f: PartialLens[U, D])
      : PartialLens[(T, U), (C, D)] =
    PartialLens {
      case (t, u) =>
        for {
          x <- apply(t)
          y <- f apply u
        } yield CoState(
          (x.get, y.get)
        , w => (x set w._1, y set w._2)
        )
    }
  // State value for monadic programming
  def state: State[T, Option[C]] =
    State(t => (apply(t) map (_.get), t))
  // Modify constructor values
  def modify(f: C => C): T => T =
    t => apply(t) match {
      case None    => t
      case Some(w) => w.set(f(w.get))
    }
}
object PartialLens {
  // Partial lens category identity
  def id[A]
      : PartialLens[A, A] =
    PartialLens(a => Some(CoState(a,
        identity)))
  // Partial lens codiagonal
  def codiag[A]
      : PartialLens[Either[A, A], A] =
    id ||| id
}
```

Given a regular lens, it is (trivially) possible to produce a partial lens by putting the `CoState` value in the `Some` constructor. That is to say, there exists a *homomorphism* from the `Lens` category to the `PartialLens` category. This is particularly useful for working with partial lenses and incorporating lenses on record types e.g. to compose with them.

Listing 33: PartialLens → Lens homomorphism

```
def homomorphism[A, B]
    : FLens[A, B] => PartialLens[A, B] =
  l =>
    PartialLens(a => Some(l apply a))
```

7. Existing Work to Integrate Lenses in Scala

Integrating lenses into Scala requires the production of lens instances for user-defined records and an accompanying library for performing operations on those lenses. Producing a rich library is simple enough without interfering with the Scala language itself, however, providing a lens value for each field of each record is less approachable.

7.1 Lensed

Lensed is a Scala compiler plugin[10] by Gerolf Seitz. It automatically inserts instances of `scalaz.Lens` on the companion object for every field of a Scala **`case class`**. The Scalaz library[7] then provides rich support for fully utilising lenses.

The obvious caveat is that Lensed is a compiler plugin, deviating from the Scala language specification. Subsequently, tools such as existing language parsers, IDEs, etc. would require modification to deal with the non-standard code. Nevertheless, the Lensed compiler plugin is quite effective at mitigating the relatively laborious effort of producing lens values and the library support provided by Scalaz gives rise to rich abstractions.

7.2 Scala Macros

Scala Macros[3] is a proposed addition to the Scala programming language providing compile-time metaprogramming. The proposed ability of Scala Macros provides the ability to automatically generate lens instances for record fields in any form (not just case classes). Since it is general, it also gives rise to the possibility of other lens implementations and representations.

Although Scala Macros appears to be the most promising for making lenses even more viable for Scala, it is a relatively new project and the implementation does not yet have the ability.

7.3 Partial Lenses

Unfortunately, there is no existing effort to incorporate partial lenses into Scala. This is despite the Scala language having very good support for sum types and the Scala library implementing many examples of same. It is hoped that improving awareness of the utility of partial lenses can inspire more investigation into this area.

7.4 Hand-rolling

Hand-rolling lens values for record fields has been a typical approach for several years, accompanied with the Scalaz library for support. It is as flexible as can be, operating on records of all types —for example, abstract algebraic data types, with hidden constructors. However, the drawback is the tedious labour required —although that labour is not to be dismissed, given the high value of abstraction and utility that lenses provide. The hand-rolling of a lens typically involves the production of boiler-plate code which we have already seen in listing 5.7:

Listing 34: The boiler-plate required for lenses

```
val field =
  Lens(
    _.field
  , (record, new_field) =>
      record copy (field = new_field)
  )
```

It could well be argued that such mechanical derivation on lenses is so trivial as to not require a language modification, especially in light of the high abstraction and utility that lenses provide as a result.

Nevertheless, aim high and higher again.

References

[1] S. Awodey. *Category theory*, volume 49. Oxford University Press, USA, 2006.

[2] A. Bohannon, B.C. Pierce, and J.A. Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347. ACM, 2006.

[3] Eugene Burmako. Scala macros. http://scalamacros.org/.

[4] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29, May 2007.

[5] J.N. Foster, A. Pilkiewicz, and B.C. Pierce. Quotient lenses. In *ACM Sigplan Notices*, volume 43, pages 383–396. ACM, 2008.

[6] M. Jones. Functional programming with overloading and higher-order polymorphism. *Advanced Functional Programming*, pages 97–136, 1995.

[7] T. Morris, R. Bjarnason, T. Adams, K. Domagala, B. Clow, R. Clarkson, P. Chiusano, T. Laugstøl, N. Partridge, J. Zaugg, E. Kmett, J. Suereth, A. Romanov, M. Hibberd, R. Nuttycombe, K. ad Wallace, G. Seitz, J. Teigen, Y. Laupa, S. Tremmel, and L. Hupel. Scalaz. http://code.google.com/p/scalaz/.

[8] Tony Morris. Source code for asymmetric lenses in scala. https://github.com/tonymorris/lenses-in-scala-code.

[9] Russell O'Connor. Functor is to lens as applicative is to biplate: Introducing multiplate. *CoRR*, abs/1103.2841, 2011.

[10] Gerolf Seitz. Lensed. https://github.com/gseitz/Lensed.