Design your own optics to improve composability and testability

YOW! Lambda Jam 2016

James Hales

Case study

We want to export a view of our very wide, sparse records, in a format supported by our downstream consumers.

Input example

```
01. val input = Map(
02.   "foo" -> Left("Apple"),
03.   "bar" -> Right(42),
04.   "baz" -> Left("Banana")
05. )
```

Output example

```
01. val output = Map(
02.   "MY_FOO" -> Left("Apple"),
03.   "MY_BAR" -> Right(42)
04. )
```

Requirements

1. Take the `String` at `"foo"` and put it in `"MY_FOO"`
2. If `"foo"` is present but not a `String`, report an error
3. Take the `Int` at `"bar"` and put it in `"MY_BAR"`
4. If `"bar"` is present but not an `Int`, report an error
5. ⋮

Practical concerns

- Record structure
- Value types
- Validation
- Conversion
- Error-reporting

We're going to ignore most of this

So...

Essentially we're getting values from input records and setting values in output records.

Optics

1. Accessors/mutators for positions in data structures
2. Laws relate accessor and mutator behaviour
3. Compose to access/mutate positions in nested data structures

Optics libraries

- Haskell — `lens` by Edward Kmett, et al.
- Scala — `Monocle` by Julien Truffaut, et al.
- Other libraries for Haskell — `data-accessor`, `data-lens`, `fclabels`, `lenses`, `lens-family`, …
- Libraries for other languages — `FSharpx.Extras`, `lens` (for Racket), `lenses.js`, `ocaml-lens`, `purescript-lens`, …

Lenses

A lens `Lens[S, A]` may be defined by two functions:

```
01. def get(s: S): A
02. def set(a: A)(s: S): S
```

Lens laws

A lens obeys the following laws:

```
01. get(set(a)(s))     == a
02. set(get(s))(s)     == s
03. set(a)(set(b)(s)) == set(a)(s)
```

for every `s: S`, `a: A`, and `b: A`

Lens examples

`first[A, B]: Lens[(A, B), A]`

Accesses/mutates the first entry in a pair.

```
("Apple", 42) applyLens first get
// "Apple"
("Apple", 42) applyLens first set "Banana"
// ("Banana", 42)
```

Lens examples

`at(k: K): Lens[Map[K, V], Option[V]]`

Accesses/mutates the value at key `k` in a map.

```
01. Map("foo" -> "Apple") applyLens at("foo") get
02. // Some("Apple")
03. Map("foo" -> "Apple") applyLens at("bar") get
04. // None
```

Lens examples

(continued)

```
01. … at("foo") set Some("Banana")
02. // Map("foo" -> "Banana")
03. … at("bar") set Some("Banana")
04. // Map("foo" -> "Apple", "bar" -> "Banana")
05. … at("foo") set None
06. // Map()
```

Lens composition

Lenses may be composed into new lenses, e.g.

```
01. first composeLens at(k):
02.   Lens[(Map[K, V], B), Option[V]]
```

Accesses/mutates the value at key `k` in the map in the first entry of a pair.

Prisms

A prism `Prism[S, A]` may be defined by two functions:

```
01. def getOption(s: S): Option[A]
02. def reverseGet(a: A): S
```

Prism laws

A prism obeys the following laws:

```
01. getOption(reverseGet(a))              == Some(a)
02. getOption(s).fold(s, reverseGet(_)) == s
```

for every `s: S`, and `a: A`

Prism examples

`stdLeft[A, B]: Prism[Either[A, B], A]`

Accesses/mutates the value in the left case of an either.

```
01. Left("Apple") applyPrism stdLeft getOption
02. // Some("Apple")
03. Right(42) applyPrism stdLeft getOption
04. // None
```

Prism examples

(continued)

```
01. "Apple" applyPrism stdLeft reverseGet
02. // Left("Apple")
```

Prism examples

`some[A]: Prism[Option[A], A]`

Accesses/mutates the value in the some case of an option.

```
Some("Apple") applyPrism some getOption
// Some("Apple")
None applyPrism some getOption
// None
```

Prism examples

(continued)

```
01. "Apple" applyPrism some reverseGet
02. // Some("Apple")
```

Prism composition

Prisms may be composed into new prisms, e.g.

```
some composePrism stdLeft:
  Prism[Option[Either[A, B]], A]
```

Accesses/mutates the value in the left case of the some case of an option of an either.

Lens/prism composition

Lenses and prisms may be composed into ???, e.g.

```
at(k) composePrism some composePrism stdLeft:
  ???[Map[K, Either[A, B]], A]
```

Lens/prism composition

Lenses and prisms may be composed into optionals, e.g.

```
at(k) composePrism some composePrism stdLeft:
  Optional[Map[K, Either[A, B]], A]
```

Or an affine Traversal in Haskell `lens`

— which is just a Traversal that accesses/mutates 0 or 1 values.

Optionals

An optional `Optional[S, A]` may be defined by two functions:

```
01. def getOption(s: S): Option[A]
02. def set(a: A)(s: S): S
```

Optional laws

An optional obeys the following laws:

```
01. getOption(set(a)(s))            == getOption(s).map(_ => a)
02. getOption(s).fold(s, set(_)(s)) == s
03. set(a)(set(b)(s))               == set(a)(s)
```

for every `s: S`, `a: A`, and `b: A`

Back to our original task...

How can optics help us?

```
01. val stringAtFoo: Optional[Map[K, Either[A, B]], A] =
02.   at("foo") composePrism some composePrism stdLeft
```

This accesses/mutates the `String` at `"foo"` in our records!

Let's try it out

```
01. val input = Map(
02.   "foo" -> Left("Apple")
03. )
04. input applyOptional stringAtFoo getOption
05. // Some("Apple")
```

It works!

Let's try it out

```
01. val input = Map()
02. input applyOptional stringAtFoo getOption
03. // None
```

Let's try it out

```
01. val input = Map(
02.   "foo" -> Right(13)
03. )
04. input applyOptional stringAtFoo getOption
05. // None
```

Hmm...

Problem

We can't distinguish between two cases:

1. The key `"foo"` is missing
2. The key `"foo"` is present, but contains a value of the wrong type.

We want to report an error if `"foo"` has a value of the wrong type.

Let's try it out some more

```
01. val input = Map(
02.   "foo" -> Left("Apple")
03. )
04. input applyOptional stringAtFoo set "Carrot"
05. // Map(
06. //  "foo" -> Left("Carrot")
07. //)
```

Let's try it out some more

```
01. val input = Map(
02.   "foo" -> Right(42)
03. )
04. input applyOptional stringAtFoo set "Carrot"
05. // Map(
06. //  "foo" -> Right(42)
07. //)
```

Let's try it out some more

```
01. val input = Map()
02. input applyOptional stringAtFoo set "Carrot"
03. // Map()
```

Another problem

We can't set the value at the key `"foo"` if:

1. The key `"foo"` is missing
2. The key `"foo"` is present, but contains a value of the wrong type.

We want to build an output record by inserting values at each of the keys.

Solutions

1. Don't use optics.
2. Use optics, but "manually" compose them.
3. Design your own optics.

Don't use optics?

- Don't have matching accessors/mutators.
- Don't relate accessors/mutators with laws.
- Don't have composable, separately testable components.

We tried it. The main codebase was alright. The tests were horrible.

Don't use optics?

```
01. def getOrError(s: Map[String, Either[String, Int]]) =
02.   s.get("foo") match {
03.     case None           => -\/("Missing value")
04.     case Some(Right(_)) => -\/("Mistyped value")
05.     case Some(Left(a))  => \/-(a)
06.   }
```

Don't use optics?

```
01. prop { (
02.     input0: Map[String, Either[String, Int]],
03.     a: String
04.   ) =>
05.     val input = input0 + ("foo" -> Left(a))
06.     getOrError(input) === \/-(a)
07. }
```

"Manually" compose optics?

- Don't use built-in composition of optics.
- Don't give optics as a result of composition.
- Don't respect laws in composition.

We tried it. It wasn't an improvement over not using optics.

"Manually" compose optics?

```
01. def getOrError(s: Map[String, Either[String, Int]]) =
02.   s applyLens at("foo") get match {
03.     case None          => -\/("Missing value")
04.     case Some(either) =>
05.       either applyPrism stdLeft getOption match {
06.         case None    => -\/("Mistyped value")
07.         case Some(v) => \/-(v)
08.       }
09. }
```

"Manually" compose optics?

```
01. prop { (
02.     input0: Map[String, Either[String, Int]],
03.     a: String
04.   ) =>
05.     val input = input0 applyLens at("foo") set
06.       (a applyPrism (some composePrism stdLeft) reverseGet)
07.     getOrError(input) === \/-(a)
08. }
```

Design your own optics

- Have matching accessors/mutators.
- Relate accessors/mutators with laws (your own laws).
- Have composable, separately testable components.

Improving our previous approaches naturally lead us here.

EPrisms

An error-reporting prism `EPrism[E, S, A]` may be defined by two functions:

```
01. def getOrError(s: S): E \/ A
02. def reverseGet(a: A): S
```

Edward Kmett might call this a co-indexed prism.

EPrism laws

We define:

```
01. def getOption(s: S): Option[A] =
02.   getOrError(s).toOption
```

Then the usual Prism laws apply.

Uplifting Prisms

Given a `Prism[S, A]` and an `e: E` we can define:

```
01. def getOrError(s: S): E \/ A =
02.   getOption(s).fold(-\/(e), \/-(_))
```

To give an `EPrism[E, S, A]`.

EPrism examples

```
01. def eStdLeft[A, B]: EPrism[String, Either[A, B], A] =
02.   stdLeft[A, B].asEPrism("Mistyped value")
```

Accesses/mutates the value in the left case of an either.

```
01. Left("Apple") applyEPrism eStdLeft getOrError
02. // \/-("Apple")
03. Right(42) applyEPrism eStdLeft getOrError
04. // -\/("Mistyped value")
```

EPrism examples

```
01. def eSome[A]: EPrism[String, Option[A], A] =
02.   some[A].asEPrism("Missing value")
```

Accesses/mutates the value in the some case of an option.

```
01. Some("Apple") applyEPrism eSome getOrError
02. // \/-("Apple")
03. None applyEPrism eSome getOrError
04. // -\/("Missing value")
```

EPrism composition

EPrisms may be composed similarly to Prisms.

```
01. ep1 composeEPrism ep2: EPrism[E1 \/ E2, ???, ???]
02. ep1 composeEPrismLeft ep2: EPrism[Option[E1], ???, ???]
03. ep1 composeEPrismRight ep2: EPrism[Option[E2], ???, ???]
04. ep1 composeEPrismMerge ep2: EPrism[E, ???, ???]
```

We just have to choose what to do with the error type.

Scopes

A scope `Scope[S, A]` may be defined by two functions:

```
01. def getOption(s: S): Option[A]
02. def put(a: A)(s: S): S
```

Scope is short for *spectroscope*.

Scope laws

A scope obeys the following laws:

```
01. getOption(put(a)(s))              == Some(a)
02. getOption(s).fold(s, put(_)(s)) == s
03. put(a)(put(b)(s))                 == put(a)(s)
```

for every `s: S`, `a: A`, and `b: A`

It's an Optional with a setter method that always succeeds.

EScopes

An error-reporting scope `EScope[E, S, A]` may be defined by two functions:

```
01. def getOrError(s: S): E \/ A
02. def put(a: A)(s: S): S
```

Similar to EPrisms.

EScope example

A slight variation of the Optional we tried earlier.

```
01. val stringAtFoo: EScope[String, Map[K, Either[A, B]], A] =
02.   at("foo") composeEPrismAsEscope
03.     eSome composeEPrismMerge eStdLeft
```

You can make a spectroscopes with lenses and prisms.

Let's try it out

```
01. val input = Map(
02.   "foo" -> Left("Apple")
03. )
04. input applyEScope stringAtFoo getOrError
05. // \/-("Apple")
```

Let's try it out

```
01. val input = Map()
02. input applyEScope stringAtFoo getOrError
03. // -\/("Missing value")
```

Let's try it out

```
01. val input = Map(
02.   "foo" -> Right(13)
03. )
04. input applyEScope stringAtFoo getOrError
05. // -\/("Mistyped value")
```

Success!

We can now distinguish between the two failure cases.

Let's try it out some more

```
01. val input = Map(
02.   "foo" -> Left("Apple")
03. )
04. input applyEScope stringAtFoo put "Carrot"
05. // Map(
06. //  "foo" -> Left("Carrot")
07. //)
```

Let's try it out some more

```
01. val input = Map(
02.   "foo" -> Right(42)
03. )
04. input applyEScope stringAtFoo put "Carrot"
05. // Map(
06. //  "foo" -> Left("Carrot")
07. //)
```

Let's try it out some more

```
01. val input = Map()
02. input applyEScope stringAtFoo put "Carrot"
03. // Map(
04. //  "foo" -> Left("Carrot")
05. //)
```

More success!

We can now set a value in a record without a value already existing.

Design your own optics

```
01. val stringAtFoo: EScope[String, Map[K, Either[A, B]], A] =
02.   at("foo") composeEPrismAsEscope
03.     eSome composeEPrismMerge eStdLeft
```

Design your own optics

```
checkAll("stringAtFoo", EScopeTests(stringAtFoo))
```

Summary

- Vanilla optics were unsuitable for our use case.
- Composition of Prisms obscures failure information.
- Composition of Lenses with Prisms weakens the setter.
- But we really wanted the benefits of optics…

Summary

- Scopes: stronger composition of Lenses and Prisms.
- EPrisms & EScopes: error-reporting Prisms and Scopes.
- Composable/interoperable with vanilla optics.
- Really easy property-based testing.
- We didn't set out to develop new optics, but we did.

Open source

Feedback and pull requests welcome.

http://github.com/CommBank/spectroscopy