LENSES:
fields as values

Scalathon 2012 • Philadelphia

Seth Tisue http://tisue.net @SethTisue

Northwestern University • Composable Solutions

July 29, 2012

this may interest you if...

...you use immutable objects

...you use *nested* immutable objects

...you want to abstract over different fields
in your immutable objects

this talk is about

functional programming

Shapeless

(type-level programming)

I am not

the author of Shapeless

Miles Sabin

≠

me

I am not a

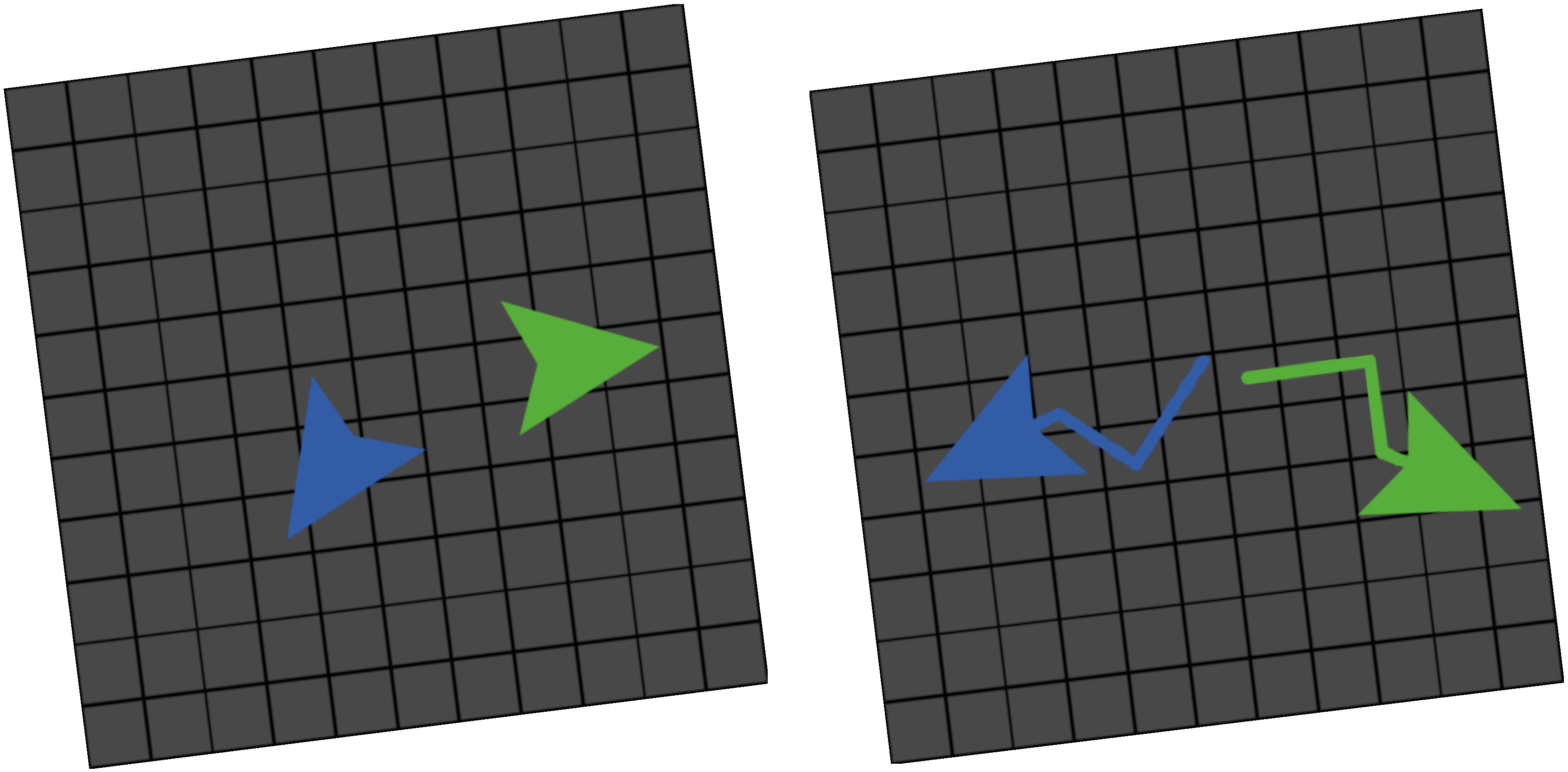
- Haskell expert
- category theorist
- Scalaz wizard

try it yourself

```
% git clone \
  https://github.com/SethTisue/lens-examples
% cd lens-examples
% cat build.sbt
libraryDependencies +=
  "com.chuusai" %% "shapeless" % "1.2.2"
% ./sbt
> test
...
> console
...
```

example domain

Turtle graphics!



everyone loves case classes

```
case class Turtle(
  xcor: Double,
  ycor: Double,
  heading: Double,
  penDown: Boolean)
```

go turtles go

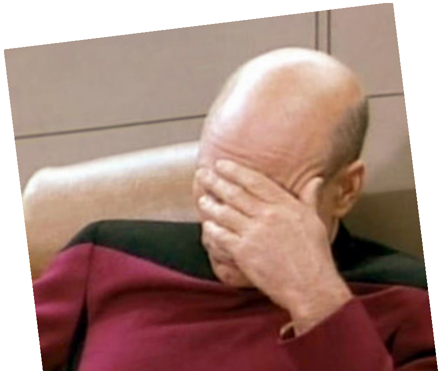
```
case class Turtle(...) {
  def right(delta: Double): Unit {
    heading += delta
  }
  def forward(dist: Double): Unit {
    xcor += dist * cos(heading)
    ycor += dist * sin(heading)
  }
  ...
```

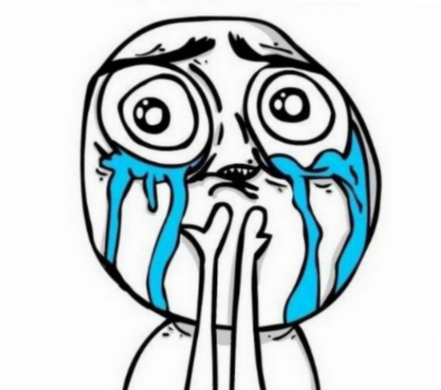
go turtles go

```
case class Turtle(
  var xcor: Double,
  var ycor: Double,
  var heading: Double,
  var penDown: Boolean)
```

we hate vars

```
case class Turtle(
  var xcor: Double,
  var ycor: Double,
  var heading: Double,
  var penDown: Boolean)
```





but we don’t need them

```
case class Turtle(
  xcor: Double,
  ycor: Double,
  heading: Double,
  penDown: Boolean)
```

~~var~~

don’t mutate — copy!

```
case class Turtle(...) {
  def right(...): Turtle =
    ...
  ...
}
```

~~var~~

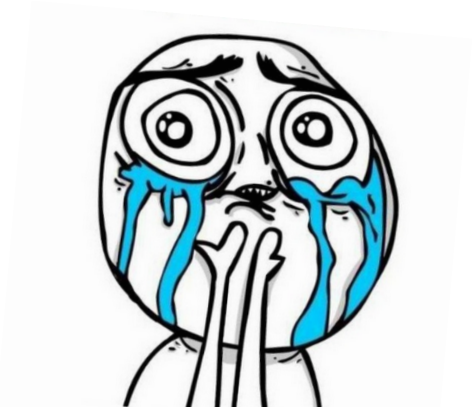
don’t mutate — copy!

```
val turtle = Turtle(...)
turtle.right(90)
```

```
val oldTurtle = Turtle(...)
val newTurtle =
  oldTurtle.right(90)
```

Ye Olde Scala (2.7)

```
case class Turtle(...) {
  def forward(...): Turtle =
    Turtle(
      xcor,
      ycor,
      heading + delta,
      penDown)
```



Scala 2.8
to the rescue

```
case class Turtle(...) {
  def forward: Turtle =
    copy(
      heading =
        heading + delta)
  ...
```

so far
so good

but

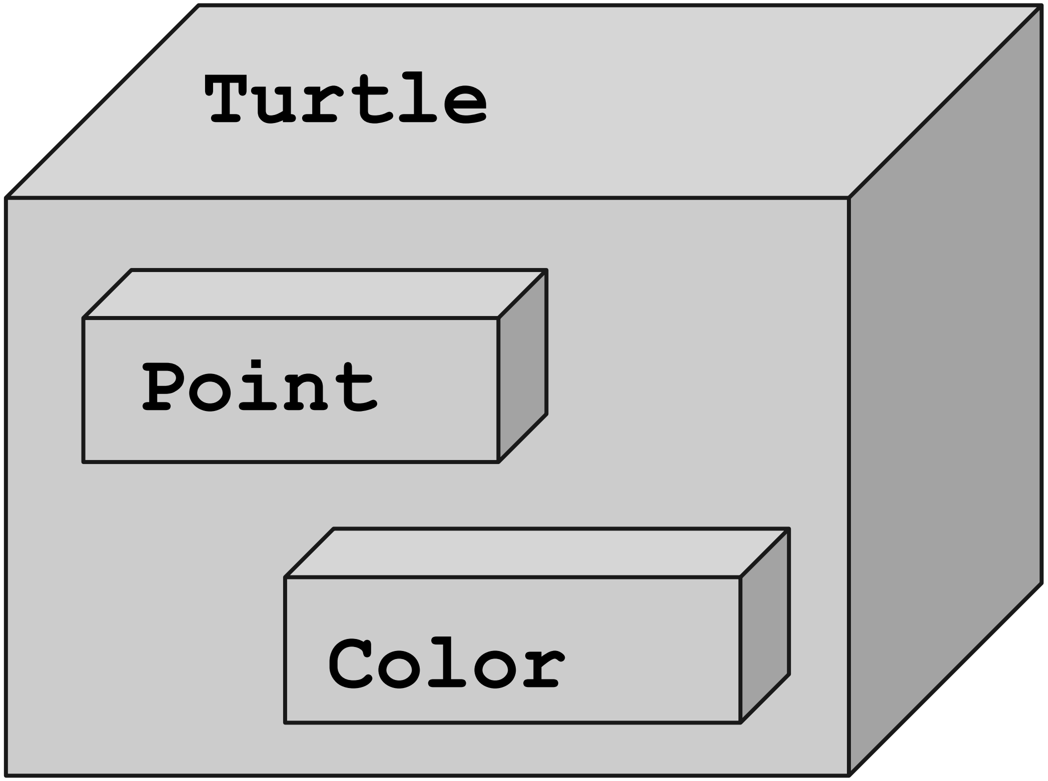
now

nesting

nesting

```
case class Point(
  x: Double, y: Double)
case class Color(
  r: Byte, g: Byte, b: Byte)
case class Turtle(
  position: Point,
  color: Color)
```

nesting



creation

```
Turtle(
  Point(2, 3),
  Color(255, 255, 255))
```

updating (mutable)

```
case class Turtle(var ..., ...) {
  def forward(dist: Double): Unit = {
    position.x += dist * cos(...)
    position.y += dist * sin(...)
  }
  ...
```

updating (immutable)

```
case class Turtle(...) {
  def forward(dist: Double): Turtle =
    copy(position =
      position.copy(
        x = position.x +
              dist * cos(...),
        y = position.y +
              dist * sin(...)))
```

it gets
worse

OO style

```
case class Turtle(...) {
  def forward(dist: Double): Turtle =
    this.copy(position =
      this.position.copy(
        x = this.position.x +
              dist * cos(this....),
        y = this.position.y +
              dist * sin(this....)))
```

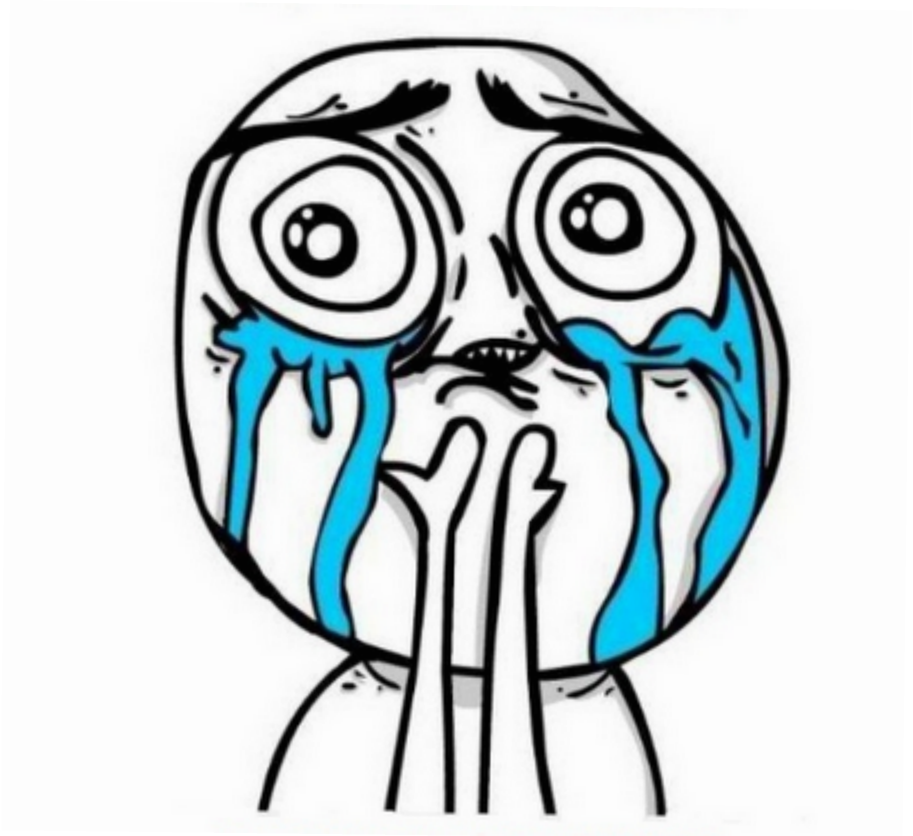
FP style

```
case class Turtle(...)  // no methods

def forward(t: Turtle, dist: Double): Turtle =
  t.copy(position =
    t.position.copy(
        x = t.position.x +
              dist * cos(t...),
        y = t.position.y +
              dist * sin(t...)))
```

worse
still

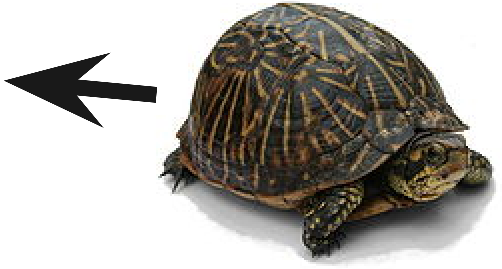
n levels deep

```
// imperative
a.b.c.d.e += 1

// functional
a.copy(
  b = a.b.copy(
    c = a.b.c.copy(
      d = a.b.c.d.copy(
        e = a.b.c.d.e +
        1))))
```



“real world”
“example

```
case class Program private(
  is3D: Boolean = false,
  interfaceGlobals: Seq[String] = Seq(),
  userGlobals: Seq[String] = Seq(),
  turtlesOwn: Seq[String] = Seq(),
  patchesOwn: Seq[String] = Seq(),
  linksOwn: Seq[String] = Seq(),
  breeds: ListMap[String, Breed] =
    ListMap(),
  linkBreeds: ListMap[String, Breed] =
    ListMap())
```



“real world”
“example

```
// if we had lenses this wouldn't get so repetitious
// - ST 7/15/12
if (isLinkBreed)
  program.copy(linkBreeds =
    orderPreservingUpdate(
      program.linkBreeds,
      program.linkBreeds(breedName).copy(
        owns = newOwns)))
else
  program.copy(breeds =
    orderPreservingUpdate(
      program.breeds,
      program.breeds(breedName).copy(
        owns = newOwns)))
```

https://github.com/NetLogo/NetLogo/blob/master/src/main/org/nlogo/compiler/StructureParser.scala#L264-275

we can fix it

omit needless repetition!

1. avoid nested `copy()`

2. abstract over similar fields

    (improving on the original imperative code)

(spoiler)

(shhhhhh... but the end result
 won’t be perfect either)

what a lens is

```
case class Lens[O, V](
  get: O => V,
  set: (O, V) => O
)
```

what a lens is

a lens is two functions:

a getter and a “setter”

what a lens is

the “setter” returns a new object

(of course!)

what a lens is

```
case class Lens[O, V](
  get: O => V,
  set: (O, V) => O
)
```

lens laws
are common sense

(0. if I get twice, I get the same answer)

1. if I get, then set it back, nothing changes.
2. if I set, then get, I get what I set.
3. if I set twice then get, I get the second thing I set.

lenses

the lens represents both things that the field can do.

in a sense, it **is** the field —
as a value

what a lens is

```
case class Lens[O, V](
  get: O => V,
  set: (O, V) => O
)
```

roll your own

```
val TurtlePosition =
  Lens[Turtle, Point](
    _.position,
   (obj, value) =>
     obj.copy(position = value))
```

roll your own

```
val PointX =
  Lens[Point, Double](
    _.x,
   (obj, value) =>
     obj.copy(x = value))
```

where it gets good

one level deep:

lens for Turtle.position

lens for Point.x

nested: combine them to get:

lens for Turtle.position.x

and so on for as many levels as you want.

the goal

```
val TurtleX =
  compose(TurtlePosition, PointX)

val t0 = Turtle()
// t0 = Turtle(Point(0.0, 0.0), ...)
val t1 = TurtleX.set(t0, 3)
// t1 = Turtle(Point(3.0, 0.0), ...)
```

composing lenses

```
def compose[Outer, Inner, Value](
      lens1: Lens[Outer, Inner],
      lens2: Lens[Inner, Value])
    : Lens[Outer, Value] =
  Lens(
    lens1.get andThen lens2.get,
    (obj, value) =>
      lens1.set(obj,
        lens2.set(lens1.get(obj),
                  value)))
```

introducing Shapeless

(abandoning our DIY lens code...)

sbt, bring me Shapeless

```
libraryDependencies +=
  "com.chuusai" %%
  "shapeless" %
  "1.2.2"
```

Shapeless, ready yourself

```
import shapeless._
import Lens._
import Nat._
```

Shapeless, study my case classes

```
implicit val pointIso =
  HListIso(Point.apply _, Point.unapply _)
implicit val colorIso =
  HListIso(Color.apply _, Color.unapply _)
implicit val turtleIso =
  HListIso(Turtle.apply _, Turtle.unapply _)
```

(we'll come back to this)

Shapeless,
BUILD ME LENSES

```
val TurtleX =
  Lens[Turtle] >> _0 >> _0
val TurtleY =
  Lens[Turtle] >> _0 >> _1
```

field numbers of Turtle of Point

the goal (again)

```
val TurtleX =
  compose(TurtlePosition, PointX)

val t0 = Turtle()
// t0 = Turtle(Point(0.0, 0.0), ...)
val t1 = TurtleX.set(t0)(3)
// t1 = Turtle(Point(3.0, 0.0), ...)
```

and better yet

```
def forward(t: Turtle,
            dist: Double) =
  TurtleY.modify(
    TurtleX.modify(t)(
      _ + dist * math.cos(t.heading)))(
        _ + dist * math.sin(t.heading))
```

abstracting over fields

```
point.copy(x = point.x + 1)
point.copy(y = point.y + 1)

// Don't Repeat Yourself
def increment(t: Turtle, ???
```

abstracting over fields

```
def increment(
    t: Turtle,
    lens: Lens[Turtle, Double]) =
  lens.modify(t)(_ + 1)

increment(t, TurtleX)
increment(t, TurtleY)
```

“real world”
“example

```
// if we had lenses this wouldn't get so repetitious
// - ST 7/15/12

if (isLinkBreed)
  program.copy(linkBreeds =
    orderPreservingUpdate(
      program.linkBreeds,
      program.linkBreeds(breedName).copy(
        owns = newOwns)))
else
  program.copy(breeds =
    orderPreservingUpdate(
      program.breeds,
      program.breeds(breedName).copy(
        owns = newOwns)))
```

“real world” “example

```
val lens =
  if (isLinkBreed)
    ProgramLinkBreedOwns
  else
    ProgramBreedOwns
lens.set(program)(newOwns)
```

higher order functions

Shapeless provides the basics:

get, set, compose, modify,
~ for lenses on pairs

additional useful stuff

Scalaz lenses provide:

/** A Lens[A,B] can be used as a function from A => B, or implicitly via Lens.asState as a State[A,B] action */

/** Modify the value viewed through the lens */

/** Modify the value viewed through the lens, a functor full of results */

/** modp[C] = modf[PartialApply1Of2[Tuple,C]#Flip], but is more convenient to think about */

/** Lenses can be composed */

/** You can apply an isomorphism to the value viewed through the lens to obtain a new lens. */

/** Two lenses that view a value of the same type can be joined */

/** Two disjoint lenses can be paired */

/** A Lens[A,B] can be used directly as a State[A,B] that retrieves the value viewed from the state */

/** We can contravariantly map the state of a state monad through a lens */

/** Contravariantly mapping the state of a state monad through a lens is a natural transformation */

/** modify the state, and return a derived value as a state monadic action. */

/** modify the portion of the state viewed through the lens and return its new value */

/** modify the portion of the state viewed through the lens, but do not return its new value */

/** Set the value viewed through the lens to a given value */

/** flatMapping a lens yields a state action to avoid ambiguity */

/** Mapping a lens yields a state action to avoid ambiguity */

/** The identity lens for a given object */

/** The trivial lens that can retrieve Unit from anything */

/** A lens that discards the choice of Right or Left from Either */

/** Access the first field of a tuple */

/** Access the second field of a tuple */

/** Lenses form a category */

/** Lenses may be used implicitly as State monadic actions that get the viewed portion of the state */

/** Lenses are an invariant functor. xmap can be used to transform a view into an isomorphic form */

/** There exists a generalized functor from Lenses to Function1, which just forgets how to set the value */

/** Enriches lenses that view tuples with field accessors */

/** A lens that views a Subtractable type can provide the appearance of in place mutation */

/** A lens that views an SetLike type can provide the appearance of in place mutation */

/** Setting the value of this lens will change whether or not it is present in the set */

/** A lens that views an immutable Map type can provide a mutable.Map-like API via State */

/** Allows both viewing and setting the value of a member of the map */

/** This lens has undefined behavior when accessing an element not present in the map! */

/** Provide the appearance of a mutable-like API for sorting sequences through a lens */

/** Provide an imperative-seeming API for stacks viewed through a lens */

/** Provide an imperative-seeming API for queues viewed through a lens */

/** Provide an imperative-seeming API for arrays viewed through a lens */

/** Allow the illusion of imperative updates to numbers viewed through a lens */

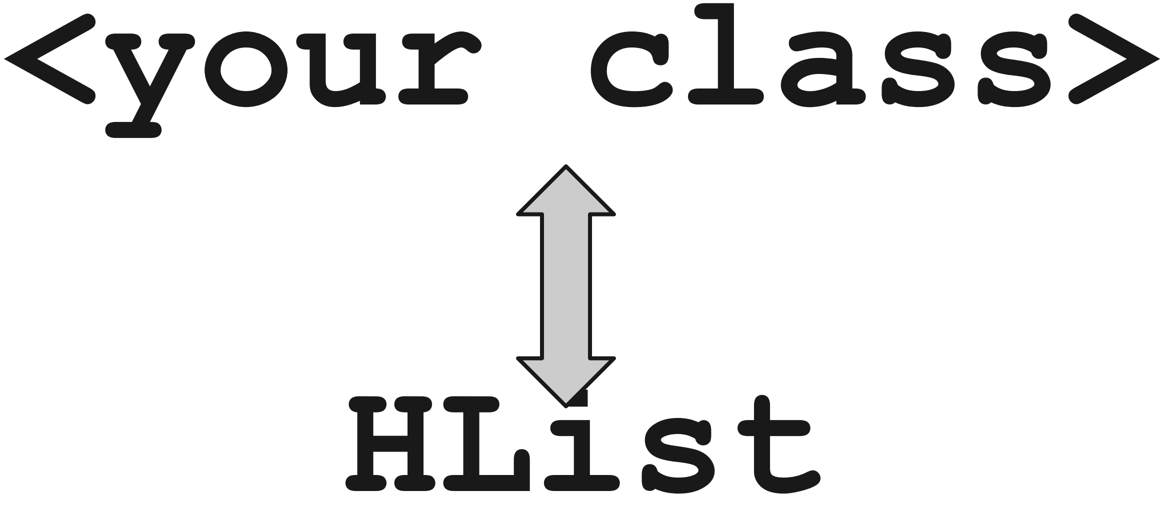
https://github.com/scalaz/scalaz/blob/master/core/src/main/scala/scalaz/Lens.scala

how it works

by what magic?

“iso” = isomorphism

```
implicit val turtleIso =
  HListIso(Turtle.apply _,
           Turtle.unapply _)
```



<your class>

HList

what’s an HList?

it’s like a tuple. each element has its own type

but tuples don’t let you *abstract over arity*

(you can’t write a function that processes tuples of any length, only a single length)

HList lifts that restriction

Shapeless, study my case classes

```
implicit val pointIso =
  HListIso(Point.apply _, Point.unapply _)
implicit val colorIso =
  HListIso(Color.apply _, Color.unapply _)
implicit val turtleIso =
  HListIso(Turtle.apply _, Turtle.unapply _)
```

some concerns

some boilerplate remains

but it’s at the definition site not use site

“unnatural” syntax and naming

depends how much immutability
is worth to you, I suppose

performance?

um, don’t ask

just the lenses please?

from Shapeless?

from Scalaz?

from Ed Kmett’s talk?

reducing boilerplate pain

Shapeless makes the boilerplate concise but doesn’t eliminate it.

macros...?

compiler plugin...? https://github.com/gseitz/Lensed/

source generation...?

source generation is not too bad

sbt makes it reasonably easy:

```
sourceGenerators in Compile <+=
  sourceManaged in Compile map { dir =>
    val file = dir / "demo" / "Test.scala"
    IO.write(file,
      """object Test extends App { println("Hi") }""")
    Seq(file)
  }
```

// TODO: write an example generator
// and add it to the GitHub repo

things I don’t know

could this be easier/better in Scala 2.10 or some future version? (type macros? untyped macros?)

how different is the Scalaz version?

how in full detail does Shapeless do it?

further viewing

Edward Kmett

"Lenses: A Functional Imperative" (2011)

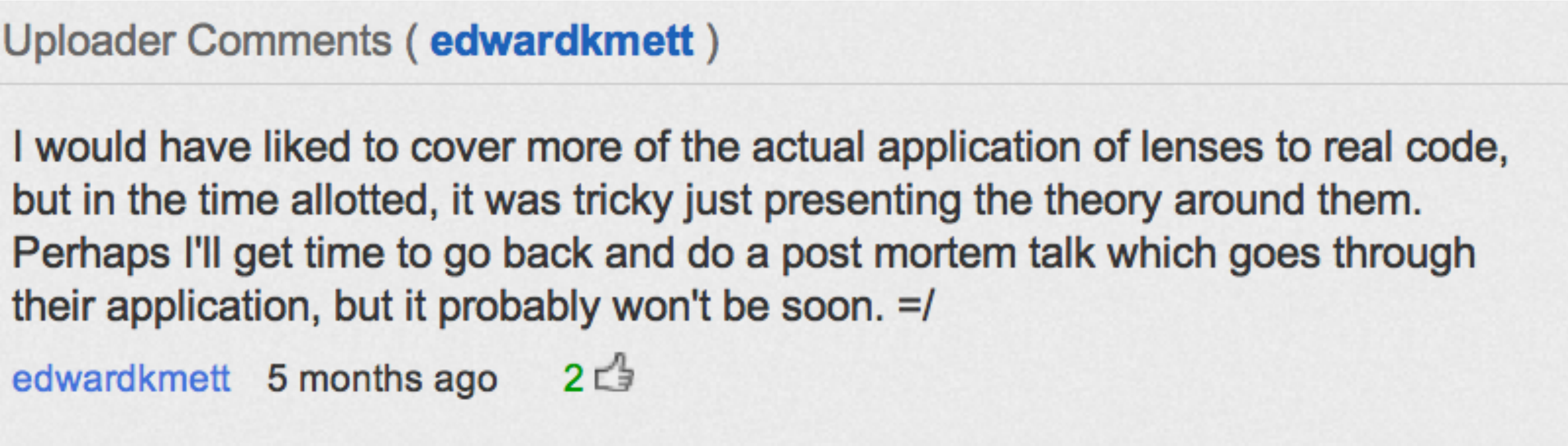
Boston Area Scala Enthusiasts

~60 minutes

lenses + state monad too

http://www.youtube.com/watch?v=efv0SQNde5Q

further viewing



Uploader Comments (**edwardkmett**)

I would have liked to cover more of the actual application of lenses to real code, but in the time allotted, it was tricky just presenting the theory around them. Perhaps I'll get time to go back and do a post mortem talk which goes through their application, but it probably won't be soon. =/

edwardkmett 5 months ago 2 👍

further viewing

Miles Sabin

“Shapeless: Exploring Generic Programming in Scala” (2012)

Northeast Scala Symposium

~30 minutes

linked from nescala.org

further reading

blog post (2012):
Jordan West,
“An Introduction
to Lenses in Scalaz”

http://www.stackmob.com/2012/02/an-introduction-to-lenses-in-scalaz/

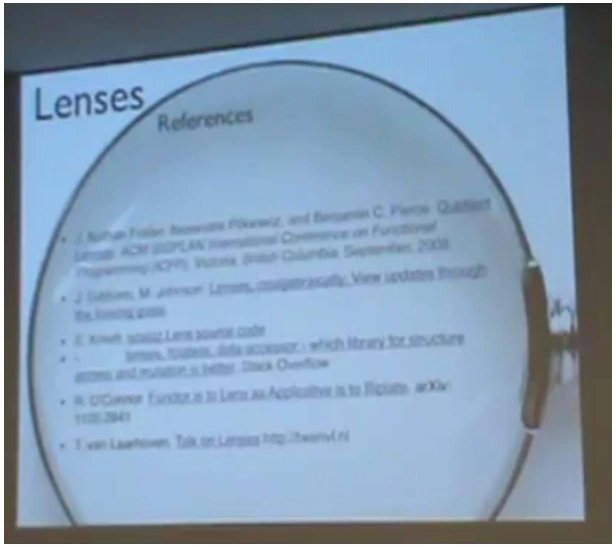
further reading

Meijer, Fokkinga & Paterson (1991)

"Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire"

http://eprints.eemcs.utwente.nl/7281/01/db-utwente-40501F46.pdf

& lots of stuff in the Haskell literature



Lenses

References

- J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient Lenses. ACM SIGPLAN International Conference on Functional Programming (ICFP), Victoria, British Columbia, September, 2008
- J. Gibbons, M. Johnson: Lenses, coalgebraically: View updates through the looking glass
- E. Kmett: scalaz Lens source code
- lenses, fclabels, data-accessor - which library for structure access and mutation is better. Stack Overflow
- R. O'Connor: Functor is to Lens as Applicative is to Biplate. arXiv 1103.2841
- T. van Laarhoven. Talk on Lenses http://twanvl.nl

questions?

easy questions:
answered on the spot

hard questions:
answered later, but *ask anyway!*

Seth Tisue • http://tisue.net • @SethTisue