package com.twitter.servo.data

import scala.language.existentials

object Lens {

private[this] val \_identity = iso[Any, Any](x => x, x => x)

/\*\*

\* The identity lens.

\*/

def identity[A] = \_identity.asInstanceOf[Lens[A, A]]

/\*\*

\* Convenience method for creating lenses with slightly more

\* efficient setters.

\*/

def checkEq[A, B](get: A => B, set: (A, B) => A) = Lens[A, B](get, set).checkEq

/\*\*

\* Create a lens from an isomorphism.

\*/

def iso[A, B](to: A => B, from: B => A) = Lens[A, B](to, (\_, x) => from(x))

/\*\*

\* Using multiple lenses, copy multiple fields from one object to another, returning

\* the updated result.

\*/

def copyAll[A](lenses: Lens[A, \_]\*)(src: A, dst: A): A =

lenses.foldLeft(dst) { (t, l) =>

l.copy(src, t)

}

/\*\*

\* setAll can be used to set multiple values using multiple lenses on the same input

\* value in one call, which is more readable than nested calls. For example, say

\* that we have lenses (lensX: Lens[A, X]), (lensY: Lens[A, Y]), and (lensZ: Lens[A, Z]),

\* then instead of writing:

\*

\* lensX.set(lensY.set(lensZ.set(a, z), y), x)

\*

\* you can write:

\*

\* Lens.setAll(a, lensX -> x, lensY -> y, lensZ -> z)

\*/

def setAll[A](a: A, lensAndValues: ((Lens[A, B], B) forSome { type B })\*): A =

lensAndValues.foldLeft(a) { case (a, (l, b)) => l.set(a, b) }

/\*\*

\* Combines two lenses into one that gets and sets a tuple of values.

\*/

def join[A, B, C](lensB: Lens[A, B], lensC: Lens[A, C]): Lens[A, (B, C)] =

Lens[A, (B, C)](

a => (lensB.get(a), lensC.get(a)),

{ case (a, (b, c)) => lensC.set(lensB.set(a, b), c) }

)

/\*\*

\* Combines three lenses into one that gets and sets a tuple of values.

\*/

def join[A, B, C, D](

lensB: Lens[A, B],

lensC: Lens[A, C],

lensD: Lens[A, D]

): Lens[A, (B, C, D)] =

Lens[A, (B, C, D)](

a => (lensB.get(a), lensC.get(a), lensD.get(a)),

{ case (a, (b, c, d)) => lensD.set(lensC.set(lensB.set(a, b), c), d) }

)

}

/\*\*

\* A Lens is a first-class getter/setter. The value of lenses is that

\* they can be composed with other operations.

\*

\* Note that it is up to you to ensure that the functions you pass to

\* Lens obey the following laws for all inputs:

\*

\* a => set(a, get(a)) == a

\* (a, b) => get(set(a, b)) == b

\* (a, b, b1) => set(set(a, b), b1) == set(a, b1)

\*

\* The intuition for the name Lens[A, B] is that you are "viewing" A

\* through a Lens that lets you see (and manipulate) a B.

\*

\* See e.g.

\* http://stackoverflow.com/questions/5767129/lenses-fclabels-data-accessor-which-library-for-structure-access-and-mutatio#answer-5769285

\* for a more in-depth explanation of lenses.

\*/

case class Lens[A, B](get: A => B, set: (A, B) => A) {

/\*\*

\* Get the field.

\*/

def apply(a: A) = get(a)

/\*\*

\* Compose with another lens, such that the setter updates the

\* outermost structure, and the getter gets the innermost structure.

\*/

def andThen[C](next: Lens[B, C]) =

Lens(get andThen next.get, (a: A, c: C) => set(a, next.set(get(a), c)))

/\*\*

\* An operator alias for `andThen`.

\*/

def >>[C](next: Lens[B, C]) = andThen(next)

/\*\*

\* Lift the function on the viewed value to a function on the outer

\* value.

\*/

def update(f: B => B): A => A = a => set(a, f(get(a)))

/\*\*

\* Copies the field from one object to another.

\*/

def copy(src: A, dst: A): A = set(dst, get(src))

/\*\*

\* Lift a mutation of the viewed value to a transform of the

\* container. (E.g. a Mutation[Seq[UrlEntity]] to a Mutation[Tweet])

\*/

def mutation(m: Mutation[B]) =

Mutation[A] { a =>

m(get(a)) map { set(a, \_) }

}

/\*\*

\* Create a new lens whose setter makes sure that the update would

\* change the value.

\*

\* This should not change the meaning of the lens, but can possibly

\* make it more efficient by avoiding copies when performing no-op

\* sets.

\*

\* This is only worthwhile when the getter and equality comparison

\* are cheap compared to the setter.

\*/

def checkEq = Lens[A, B](get, (a, b) => if (get(a) == b) a else set(a, b))

/\*\*

\* Combines this lens and the given lens into one that gets and sets a tuple

\* of values.

\*/

def join[C](right: Lens[A, C]): Lens[A, (B, C)] =

Lens.join(this, right)

}