package com.twitter.servo.data

import com.twitter.util.{Return, Throw, Try}

import com.twitter.finagle.stats.{Counter, StatsReceiver}

import com.twitter.servo.util.{Effect, Gate}

object Mutation {

/\*\*

\* A mutation that ignores its input and always returns the given

\* value as new. Use checkEq if this value could be the same as the

\* input.

\*/

def const[T](x: T) = Mutation[T] { \_ =>

Some(x)

}

private[this] val \_unit = Mutation[Any] { \_ =>

None

}

/\*\*

\* A "no-op" mutation that will never alter the value.

\*

\* For any Mutations A, (A also unit) == (unit also A) == A.

\*

\* Forms a monoid with also as the operation.

\*/

def unit[A]: Mutation[A] = \_unit.asInstanceOf[Mutation[A]]

/\*\*

\* Makes a Mutation out of a function.

\*/

def apply[A](f: A => Option[A]): Mutation[A] =

new Mutation[A] {

override def apply(x: A) = f(x)

}

/\*\*

\* Lift a function that returns the same type to a Mutation, using

\* the type's notion of equality to detect when the mutation has

\* not changed the value.

\*/

def fromEndo[A](f: A => A): Mutation[A] =

Mutation[A] { x =>

val y = f(x)

if (y == x) None else Some(y)

}

/\*\*

\* Lift a partial function from A to A to a mutation.

\*/

def fromPartial[A](f: PartialFunction[A, A]): Mutation[A] = Mutation[A](f.lift)

/\*\*

\* Creates a new Mutation that applies all the given mutations in order.

\*/

def all[A](mutations: Seq[Mutation[A]]): Mutation[A] =

mutations.foldLeft(unit[A])(\_ also \_)

}

/\*\*

\* A Mutation encapsulates a computation that may optionally "mutate" a value, where

\* "mutate" should be interpreted in the stateless/functional sense of making a copy with a

\* a change. If the value is unchanged, the mutation should return None. When mutations are

\* composed with `also`, the final result will be None iff no mutation actually changed the

\* value.

\*

\* Forms a monoid with Mutation.unit as unit and `also` as the

\* combining operation.

\*

\* This abstraction is useful for composing changes to a value when

\* some action (such as updating a cache) should be performed if the

\* value has changed.

\*/

trait Mutation[A] extends (A => Option[A]) {

/\*\*

\* Convert this mutation to a function that always returns a

\* result. If the mutation has no effect, it returns the original

\* input.

\*

\* (convert to an endofunction on A)

\*/

lazy val endo: A => A =

x =>

apply(x) match {

case Some(v) => v

case None => x

}

/\*\*

\* Apply this mutation, and then apply the next mutation to the

\* result. If this mutation leaves the value unchanged, the next

\* mutation is invoked with the original input.

\*/

def also(g: Mutation[A]): Mutation[A] =

Mutation[A] { x =>

apply(x) match {

case None => g(x)

case someY @ Some(y) =>

g(y) match {

case some @ Some(\_) => some

case None => someY

}

}

}

/\*\*

\* Apply this mutation, but refuse to return an altered value. This

\* yields all of the effects of this mutation without affecting the

\* final result.

\*/

def dark: Mutation[A] = Mutation[A] { x =>

apply(x); None

}

/\*\*

\* Convert a Mutation on A to a Mutation on B by way of a pair of functions for

\* converting from B to A and back.

\*/

def xmap[B](f: B => A, g: A => B): Mutation[B] =

Mutation[B](f andThen this andThen { \_ map g })

/\*\*

\* Converts a Mutation on A to a Mutation on Try[A], where the Mutation is only applied

\* to Return values and any exceptions caught by the underying function are caught and

\* returned as Some(Throw(\_))

\*/

def tryable: Mutation[Try[A]] =

Mutation[Try[A]] {

case Throw(x) => Some(Throw(x))

case Return(x) =>

Try(apply(x)) match {

case Throw(y) => Some(Throw(y))

case Return(None) => None

case Return(Some(y)) => Some(Return(y))

}

}

/\*\*

\* Perform this mutation only if the provided predicate returns true

\* for the input.

\*/

def onlyIf(predicate: A => Boolean): Mutation[A] =

Mutation[A] { x =>

if (predicate(x)) this(x) else None

}

/\*\*

\* Performs this mutation only if the given gate returns true.

\*/

def enabledBy(enabled: Gate[Unit]): Mutation[A] =

enabledBy(() => enabled())

/\*\*

\* Performs this mutation only if the given function returns true.

\*/

def enabledBy(enabled: () => Boolean): Mutation[A] =

onlyIf { \_ =>

enabled()

}

/\*\*

\* A new mutation that returns the same result as this mutation,

\* and additionally calls the specified Effect.

\*/

def withEffect(effect: Effect[Option[A]]): Mutation[A] =

Mutation[A](this andThen effect.identity)

/\*\*

\* Perform an equality check when a value is returned from the

\* mutation. If the values are equal, then the mutation will yield

\* None.

\*

\* This is useful for two reasons:

\*

\* 1. Any effects that are conditional upon mutation will not occur

\* when the values are equal (e.g. updating a cache)

\*

\* 2. When using a Lens to lift a mutation to a mutation on a

\* larger structure, checking equality on the smaller structure

\* can prevent unnecessary copies of the larger structure.

\*/

def checkEq = Mutation[A] { x =>

this(x) match {

case someY @ Some(y) if y != x => someY

case \_ => None

}

}

/\*\*

\* Converts this mutation to a mutation of a different type, using a Lens to

\* convert between types.

\*/

def lensed[B](lens: Lens[B, A]): Mutation[B] =

Mutation[B](b => this(lens(b)).map(lens.set(b, \_)))

/\*\*

\* Convert this mutation to a mutation of a Seq of its type. It will

\* yield None if no values are changed, or a Seq of both the changed

\* and unchanged values if any value is mutated.

\*/

def liftSeq = Mutation[Seq[A]] { xs =>

var changed = false

val detectChange = Effect.fromPartial[Option[A]] { case Some(\_) => changed = true }

val mutated = xs map (this withEffect detectChange).endo

if (changed) Some(mutated) else None

}

/\*\*

\* Convert this mutation to a mutation of a Option of its type. It will yield

\* None if the value is not changed, or a Some(Some(\_)) if the value is mutated.

\*/

def liftOption = Mutation[Option[A]] {

case None => None

case Some(x) =>

this(x) match {

case None => None

case Some(y) => Some(Some(y))

}

}

/\*\*

\* Convert this mutation to a mutation of the values of a Map. It will

\* yield None if no values are changed, or a Map with both the changed

\* and unchanged values if any value is mutated.

\*/

def liftMapValues[K] = Mutation[Map[K, A]] { m =>

var changed = false

val detectChange = Effect.fromPartial[Option[A]] { case Some(\_) => changed = true }

val f = (this withEffect detectChange).endo

val mutated = m map { case (k, v) => (k, f(v)) }

if (changed) Some(mutated) else None

}

/\*\*

\* Return a new mutation that returns the same result as this

\* mutation, as well as incrementing the given counter when the

\* value is mutated.

\*/

def countMutations(c: Counter) =

this withEffect { Effect.fromPartial { case Some(\_) => c.incr() } }

/\*\*

\* Wrap a mutation in stats with the following counters:

\* - no-op (returned value was the same as the input)

\* - none (mutation returned none)

\* - mutated (mutation modified the result)

\*/

def withStats(stats: StatsReceiver): Mutation[A] = {

val none = stats.counter("none")

val noop = stats.counter("noop")

val mutated = stats.counter("mutated")

input: A => {

val result = apply(input)

result.fold(none.incr()) { output =>

if (output == input) {

noop.incr()

} else {

mutated.incr()

}

}

result

}

}

}