package com.twitter.servo.util

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.util.{Duration, Time}

import java.util.concurrent.ThreadLocalRandom

import scala.language.implicitConversions

object Gate {

/\*\*

\* Construct a new Gate from a boolean function and a string representation

\*/

def apply[T](f: T => Boolean, repr: => String): Gate[T] =

new Gate[T] {

override def apply[U](u: U)(implicit asT: <:<[U, T]): Boolean = f(asT(u))

override def toString: String = repr

}

/\*\*

\* Construct a new Gate from a boolean function

\*/

def apply[T](f: T => Boolean): Gate[T] = Gate(f, "Gate(" + f + ")")

/\*\*

\* Create a Gate[Any] with a probability of returning true

\* that increases linearly with the availability, which should range from 0.0 to 1.0.

\*/

def fromAvailability(

availability: => Double,

randomDouble: => Double = ThreadLocalRandom.current().nextDouble(),

repr: String = "Gate.fromAvailability"

): Gate[Any] =

Gate(\_ => randomDouble < math.max(math.min(availability, 1.0), 0.0), repr)

/\*\*

\* Creates a Gate[Any] with a probability of returning true that

\* increases linearly in time between startTime and (startTime + rampUpDuration).

\*/

def linearRampUp(

startTime: Time,

rampUpDuration: Duration,

randomDouble: => Double = ThreadLocalRandom.current().nextDouble()

): Gate[Any] = {

val availability = availabilityFromLinearRampUp(startTime, rampUpDuration)

fromAvailability(

availability(Time.now),

randomDouble,

repr = "Gate.rampUp(" + startTime + ", " + rampUpDuration + ")"

)

}

/\*\*

\* Generates an availability function that maps a point in time to an availability value

\* in the range of 0.0 - 1.0. Availability is 0 if the given time is before startTime, is

\* 1 if the greather than (startTime + rampUpDuration), and is otherwise linearly

\* interpolated between 0.0 and 1.0 as the time moves through the two endpoints.

\*/

def availabilityFromLinearRampUp(startTime: Time, rampUpDuration: Duration): Time => Double = {

val endTime = startTime + rampUpDuration

val rampUpMillis = rampUpDuration.inMilliseconds.toDouble

now => {

if (now >= endTime) {

1.0

} else if (now <= startTime) {

0.0

} else {

(now - startTime).inMilliseconds.toDouble / rampUpMillis

}

}

}

/\*\*

\* Returns a gate that increments true / false counters for each Gate invocation. Counter name

\* can be overridden with trueName and falseName.

\*/

def observed[T](

gate: Gate[T],

stats: StatsReceiver,

trueName: String = "true",

falseName: String = "false"

): Gate[T] = {

val trueCount = stats.counter(trueName)

val falseCount = stats.counter(falseName)

gate

.onTrue[T] { \_ =>

trueCount.incr()

}

.onFalse[T] { \_ =>

falseCount.incr()

}

}

/\*\*

\* Construct a new Gate from a boolean value

\*/

def const(v: Boolean): Gate[Any] = Gate(\_ => v, v.toString)

/\*\*

\* Constructs a new Gate that returns true if any of the gates in the input list return true.

\* Always returns false when the input list is empty.

\*/

def any[T](gates: Gate[T]\*): Gate[T] = gates.foldLeft[Gate[T]](Gate.False)(\_ | \_)

/\*\*

\* Constructs a new Gate that returns true iff all the gates in the input list return true.

\* Always returns true when the input list is empty.

\*/

def all[T](gates: Gate[T]\*): Gate[T] = gates.foldLeft[Gate[T]](Gate.True)(\_ & \_)

/\*\*

\* Gates that always return true/false

\*/

val True: Gate[Any] = const(true)

val False: Gate[Any] = const(false)

// Implicit conversions to downcast Gate to a plain function

implicit def gate2function1[T](g: Gate[T]): T => Boolean = g(\_)

implicit def gate2function0(g: Gate[Unit]): () => Boolean = () => g(())

}

/\*\*

\* A function from T to Boolean, composable with boolean-like operators.

\* Also supports building higher-order functions

\* for dispatching based upon the value of this function over values of type T.

\* Note: Gate does not inherit from T => Boolean in order to enforce correct type checking

\* in the apply method of Gate[Unit]. (Scala is over eager to convert the return type of

\* expression to Unit.) Instead, an implicit conversion allows Gate to be used in methods that

\* require a function T => Boolean.

\*/

trait Gate[-T] {

/\*\*

\* A function from T => boolean with strict type bounds

\*/

def apply[U](u: U)(implicit asT: <:<[U, T]): Boolean

/\*\*

\* A nullary variant of apply that can be used when T is a Unit

\*/

def apply()(implicit isUnit: <:<[Unit, T]): Boolean = apply(isUnit(()))

/\*\*

\* Return a new Gate which applies the given function and then calls this Gate

\*/

def contramap[U](f: U => T): Gate[U] = Gate(f andThen this, "%s.contramap(%s)".format(this, f))

/\*\*

\* Returns a new Gate of the requested type that ignores its input

\*/

def on[U](implicit isUnit: <:<[Unit, T]): Gate[U] = contramap((\_: U) => ())

/\*\*

\* Returns a new Gate which returns true when this Gate returns false

\*/

def unary\_! : Gate[T] = Gate(x => !this(x), "!%s".format(this))

/\*\*

\* Returns a new Gate which returns true when both this Gate and other Gate return true

\*/

def &[U <: T](other: Gate[U]): Gate[U] =

Gate(x => this(x) && other(x), "(%s & %s)".format(this, other))

/\*\*

\* Returns a new Gate which returns true when either this Gate or other Gate return true

\*/

def |[U <: T](other: Gate[U]): Gate[U] =

Gate(x => this(x) || other(x), "(%s | %s)".format(this, other))

/\*\*

\* Returns a new Gate which returns true when return values of this Gate and other Gate differ

\*/

def ^[U <: T](other: Gate[U]): Gate[U] =

Gate(x => this(x) ^ other(x), "(%s ^ %s)".format(this, other))

/\*\*

\* Returns the first value when this Gate returns true, or the second value if it returns false.

\*/

def pick[A](t: T, x: => A, y: => A): A = if (this(t)) x else y

/\*\*

\* A varient of pick that doesn't require a value if T is a subtype of Unit

\*/

def pick[A](x: => A, y: => A)(implicit isUnit: <:<[Unit, T]): A = pick(isUnit(()), x, y)

/\*\*

\* Returns a 1-arg function that dynamically picks x or y based upon the function arg.

\*/

def select[A](x: => A, y: => A): T => A = pick(\_, x, y)

/\*\*

\* Returns a version of this gate that runs the effect if the gate returns true.

\*/

def onTrue[U <: T](f: U => Unit): Gate[U] =

Gate { (t: U) =>

val v = this(t)

if (v) f(t)

v

}

/\*\*

\* Returns a version of this gate that runs the effect if the gate returns false.

\*/

def onFalse[U <: T](f: U => Unit): Gate[U] =

Gate { (t: U) =>

val v = this(t)

if (!v) f(t)

v

}

}