```
{-# LANGUAGE TypeFamilies , UndecidableInstances , FlexibleContexts #-}
module Parcial2.Labyrinth where
  import Control.Exception
  import Control.Arrow (first, second)
  import Control.Monad.Fix
  import Data.Tuple (swap)
  import Data.Maybe (isJust, fromJust)
  import Data.Set (Set, member)
  import qualified Data.Set as Set
  import GHC.Real (infinity)
  import Parcial2.ReadLabyrinth
  import GeneticAlgorithm
  import System.Random
```

1. Introducción

El mapa (laberinto), descrito en la tarea, se define como un grafo: nodos — un conjunto de puntos (con posiciones correspondientes); aristas — la existencia de rutas directas.

```
data Labyrinth point = Labyrinth {
  nodes :: Set point,
  edges :: Set (point, point),
  initial :: point,
  target :: point
  }
edgeOf p es = any (`member` es) [p, swap p]
```

Se define la *distancia directa* entre los nodos que están connectados por una arista.

```
data DirectDistance point dist = DirectDistance {
  labyrinthDist :: Labyrinth point → point → point → Maybe dist
  }
mkDirectDistance f = DirectDistance $ λl v1 v2 →
  if (v1, v2) `edgeOf` edges l then Just (f v1 v2) else Nothing
```

El algoritmo genético abstracto está definido en src/GeneticAlgorithm.hs. Su implementación se presentará adelante.

2. Implementación

2.1. Lectura de mapas

Se utiliza un mapa 2D:

```
newtype Point2D = Point2D (Int, Int) deriving Eq
pnt2D (Point2D p) = p


type Labyrinth2D = Labyrinth Point2D
```

La lectura del archivo del mapa se encuentra en src/Parcial2/ReadLabyrinth.hs. Aquí se presenta la construcción del grafo a partir del mapa leido.

La instancia de *Ord* usada determinará

```
readLabyrinth2D :: (Ord Point2D) ⇒
  FilePath → IO (Either [String] Labyrinth2D)
readLabyrinth2D file = build <$> try (readFile file)
  where
    build (Left err) = Left [displayException (err :: SomeException)]
    build (Right s) = case parseLabyrinth s of
                        Left errS → Left errS
                        Right l → Right (build' l)
    build' (LabyrinthDescription n conn (i, t) coords) =
      let get = Point2D ∘ (coords!!)
      in Labyrinth
        (Set.fromList $ map Point2D coords)
        (Set.fromList $ map (first get ∘ second get) coords)
        (get i)
        (get t)
```

2.2. Algoritmo genético

```
data GA = GA Labyrinth2D
```

Un alias para tuple `(a,a)`.

```
newtype Pair a = Pair (a, a)
unwrapPair (Pair p) = p
pair2List (Pair (f, s)) = [f, s]
```

Se define el orden **ascendiente** soble los puntos, para que los mejores cromosomas sean en el principio de la lista que representa la población.

```
instance Ord Point2D where
  compare (Point2D p1) (Point2D p2) = compare p1 p2
```

Se define la metrica sobre el grafo:

$$\text{dist}(p_1, p_2) = \begin{cases} \mathit{Just}\ d_E(p_1, p_2) & \text{si } \exists \text{ arista, connectando } p_1 \text{ y } p_2 \\ \mathit{Nothing} & \text{en otro caso} \end{cases}, \text{ donde}$$

d_E — es la distancia euclidiana entre dos puntos.

Un *gene* se define como nodo del laberinto y un *cromosoma* como una lista de genes.

```
type Gene GA = Point2D
type Chromosome GA = [Point2D]
  -- listGenes :: Chromosome ga → [Gene ga]
listGenes = id
```

Los valores de aptitud de adaptación van a tener un tipo flotante de doble precision.

```
type Fitness GA = Double
```

Para denotar que la operación de *crossover* preserva el tamaño de población, su resultado se marca como un par de hijos.

```
type CrossoverChildren GA = Pair
```

La información de entrada para generación de la población — el laberinto.

```
type InputData GA = Labyrinth2D
```

El resultado es la mejor chromosoma obtenida.

```
type ResultData GA = Chromosome GA
```

Generación de cromosomas aleatorios.

```
  -- randomChromosome :: ga → IO (Chromosome ga)
randomChromosome (GA l) = do
```

Primero se genera aleotoriamente el tamaño extra del cromosoma con valor entre 0 y $2N$. Dos valores mas se reservan para el punto inicial y el punto final. El tamaño final de los cromosomas generados está entre 2 y $2N + 2$.

```
  len' ← getStdRandom $ randomR (0, 2 * Set.size (nodes l))
```

Un punto aleatorio se selecciona entre todos los nodos del mapa, excepto la posición inicial del agente y el punto meta.

```
let randPoint = ⊥ :: StdGen → (Point2D, StdGen)
```

Un punto aleatorio se re-genera hasta que se encuentra uno que todavía no está en el cromosoma, generado previamente (ulilizando el mismo generador).

```
let rand prev = fix $ λf g →
  let (r, g') = randPoint g
  in if r ∈ prev then f g' else (r, g')
```

Se genera la parte aleatoria del cromosoma.

```
rnd ← getStdGen
let (genes, _) = ($([], rnd)) ∘ fix $
  λf (l, g) → if length l ≡ len'
    then (l, g)
    else first (:l) (rand l g)
```

Todas las rutas, encodificadas en los cromosomas, se empiezan en el punto inicial y se terminan en el punto meta.

```
let chrom = [initial l] ++ genes ++ [target l]
return chrom
```

La **aptitud de adoptación** se define como:

$$f(c) = \begin{cases} \sum_{i=1}^{\text{len}-1} \text{dist}(c_{i-1}, c_i) & \text{si} \quad \forall i = \overline{[1, \text{len}-1]} \Rightarrow \\ & \quad \exists \text{ arista, connectando } c_{i-1} \text{ y } c_i \\ +\infty & \text{en otro caso} \end{cases}$$

```
  -- fitness :: ga → Chromosome ga → Fitness ga
fitness (GA l) genes = let
  lPairs (f : s : t) = (f, s) : lPairs (s : t)
  lPairs _ = []
  dists = map (uncurry $ eDist l) (lPairs genes)
  in if isJust `all` dists
    then sum $ map fromJust dists
    else fromRational infinity
```

?

```
-- crossover :: Chromosome ga → Chromosome ga
--           → CrossoverChildren ga (Chromosome ga)
```

?

```
-- mutate :: Chromosome ga → Chromosome ga
```

?

```
-- stopCriteria :: [Fitness ga] → Bool
```

?

```
-- newGA :: InputData ga → ga
```