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

1. Introducción

El mapa (laberinto), descrito en la tarea, se defina 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 generico abstracto está definido en src/GeneticAlgorithm.hs. Su implementación se presentará adelante.

2. Implementación

2.1. Lectura de mapas

Se utiliza un mapa 2D:

```
type Point2D = (Int, Int)
type Labyrinth2D = Labyrinth Point2D
```

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

```
readLabyrinth2D :: 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 = (coords!!)
      in Labyrinth
         (Set.fromList 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 la instancia de la clase *GeneticAlgorithm* para *GA* empiezando con los tipos y siguiendo con los métodos.

```
instance GeneticAlgorithm GA where
```

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

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

- Los valores 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 población inicial.

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

Primero se genera aleotoriamente el tamaño extra de la cromosoma con valor entre 0 y $2N$. Dos valores mas se reservan para el punto inicial y el punto final. El tamaño final de las cromosomas generadas 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)
```

Se generan los puntos aleatorios hasta que se encuentra uno que todavía no está en la cromosoma, generada previomente.

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

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

⊥
```

- ?

```
-- fitness :: Chromosome ga → Fitness ga
```

- ?

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

- ?

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

- ?

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

- ?

```
-- newGA :: InputData ga → ga
```