Generales del Estudiante

Daniel Reynel Dominguez Ceballos

C-411

Marco General

El ambiente en el cual intervienen los agentes es discreto y tiene la forma de un rectángulo de N × M. El ambiente es de información completa, por tanto todos los agentes conocen toda la información sobre el ambiente. El ambiente puede variar aleatoriamente cada t unidades de tiempo. El valor de t es conocido.

Las acciones que realizan los agentes ocurren por turnos. En un turno, los agentes realizan sus acciones, una sola por cada agente, y modifican el medio sin que este varie a no ser que cambie por una accion de los agentes. En el siguiente, el ambiente puede variar.Si es el momento de cambio del ambiente, ocurre primero el cambio natural del ambiente y luego la variación aleatoria. En una unidad de tiempo ocurren el turno del agente y el turno de cambio del ambiente.

Los elementos que pueden existir en el ambiente son obstáculos, suciedad, niños, el corral y los agentes que son llamados Robots de Casa. A continuación se precisan las características de los elementos del ambiente:

Obstáculos:

Estos ocupan una única casilla en el ambiente. Ellos pueden ser movidos, empujándolos, por los niños, una única casilla. El Robot de Casa sin embargo no puede moverlo. No pueden ser movidos ninguna de las casillas ocupadas por cualquier otro elemento del ambiente.

Suciedad:

La suciedad es por cada casilla del ambiente. Solo puede aparecer en casillas que previamente estuvieron vacías. Esta, o aparece en el estado inicial o es creada por los niños

Corral:

el corral ocupa casillas adyacentes en número igual al del total de niños presentes en el ambiente. El corral no puede moverse. En una casilla del corral solo puede coexistir un niño. En una casilla del corral, que esté vacía, puede entrar un robot. En una misma casilla del corral pueden coexistir un niño y un robot solo si el robot lo carga, o si acaba de dejar al niño.

Niño:

Los niños ocupan solo una casilla. Ellos en el turno del ambiente se mueven, si es posible (si la casilla no está ocupada: no tiene suciedad, no está el corral, no hay un Robot de Casa), y aleatoriamente (puede que no ocurra movimiento), a una de las casilla adyacentes. Si esa casilla está ocupada por un obstáculo este es empujado por el niño, si en la dirección hay más de un obstáculo, entonces se desplazan todos. Si el obstáculo está en una posición donde no puede ser empujado y el niño lo intenta, entonces el obstáculo no se mueve y el niño ocupa la misma posición.

Los niños son los responsables de que aparezca suciedad. Si en una cuadrícula de 3 por 3 hay un solo niño, entonces, luego de que él se mueva aleatoriamente, una de las casillas de la cuadrícula anterior que esté vacía puede haber sido ensuciada. Si hay dos niños se pueden ensuciar hasta 3. Si hay tres niños o más pueden resultar sucias hasta 6.

Los niños cuando están en una casilla del corral, ni se mueven ni ensucian.

Si un niño es capturado por un Robot de Casa tampoco se mueve ni ensucia.

Objetivo:

El objetivo del Robot de Casa es mantener la casa limpia. Se considera la casa limpia si el 60 % de las casillas vacías no están sucias.

Robot de Casa:

El Robot de Casa se encarga de limpiar y de controlar a los niños. El Robot se mueve a una de las casillas adyacentee, las que decida. Solo se mueve una casilla sino carga un niño. Si carga un niño pude moverse hasta dos casillas consecutivas. También puede realizar las acciones de limpiar y cargar niños. Si se mueve a una casilla con suciedad, en el próximo turno puede decidir limpiar o moverse. Si se mueve a una casilla donde está un niño, inmediatamente lo carga. En ese momento, coexisten en la casilla Robot y niño. Si se mueve a una casilla del corral que está vacía, y carga un niño, puede decidir si lo deja esta casilla o se sigue moviendo. El Robot puede dejar al niño que carga en cualquier casilla. En ese momento cesa el movimiento del Robot en el turno, y coexisten hasta el próximo turno, en la misma casilla, Robot y niño.

Como correr el programa:

Para correr el programa se debe abrir una consola de ghci en la carpeta raíz y cargar $main.hs$ a continuación un ejemplo usando stack

```
Program_Root > stack ghci
>:l main.hs
```

Luego se llama a la función $main$ para correr el programa:

```
>main
```

A partir de aquí se presentarán los datos que se deben llenar para generar los ambientes sobre los cuales se realizarán las simulaciones.

Modelos de Agentes Considerados

Se utilizaron en la realización de las simulaciones 4 tipos de agentes distintos con una 5ta variante que mezcla agentes para distribuir los trabajos de cada uno. Como el ambiente es accesible el agentes se basan en percepciones para realizar las acciones que se proponen en base a objetivos iniciales que se plantean, por tanto son una combinación de reactivo con proactivo. Estos agentes tienen estados, que cambian según las percepciones y acciones que realizan. El objetivo principal de cada agente es mantener un estado, en este caso que la limpieza se mantenga por encima del 60% y mientras cumplen unos objetivos secundarios que difieren de un agente a otro, que se verán reflejados en las conductas de cada agente. Todos los agentes usan

una arquitectura de Brooks debido a que la toma de decisiones es de pocas capas y no se toman decisiones a largo plazo.

Objetivos:

- El objetivo principal de los agentes es que el estado de limpieza del ambiente no disminuya del 60%

Condición de victoria:

Se dice que una simulación alcanzó la condición de victoria cuando ocurra una de dos condiciones:

- Todos los niños están atrapados
- Después de 100 actualizaciones del ambiente (Solo en caso de que no se realice una simulación simple)

Se dice que un niño está atrapado si se encuentra en el corral o en manos de un robot. Sea ac la cantidad de niños en el corral, ar la cantidad de niños en manos de robot y n la cantidad de niños. Si $ac + ar >= n$ entonces la suciedad del tablero no podrá aumentar más, pues ningún niño podrá moverse y por tanto no podrán generar suciedad, lo que implica que el % de suciedad del ambiente no aumentará.

La segunda condición está dada para poder terminar la simulación en algún punto y que esta no entre en un ciclo infinito del ambiente (ya que no sabemos si en algún momento se fracasará o se obtendrá una condición de derrota). Por tanto se considera que si un robot logró cumplir sus objetivos durante 100 cambios que hubo en el ambiente, este será victorioso.

Condición de derrota:

Se dice que una simulación alcanzó la condición de derrota cuando el ambiente en una actualización pasa a sobrepasar el 40% de suciedad (O sea el estado de limpieza baja del 60%).

Estados:

Los agentes constan de 2 estados, en la sección de acciones se especificarán los cambios de estado que ocurren al realizar determinada acción

$Estado\ 1$: Libre

$Estado\ 2$: Cargando Niño

Percepciones:

- $availableChild$: El agente puede percibir a los niños alcanzables desde su posición si existe un camino hasta ese niño.
- $availableDirty$: El agente puede percibir a las suciedades alcanzables desde su posición si existe un camino hasta esa suciedad.
- $availableCorral$: El agente puede percibir si existe un corral alcanzable desde su posición y este se encuentre vacío, o sea que existe un camino hasta el corral .
- $sameSpotChild$: Estoy en la misma posición de un niño. Si el agente realiza un movimiento hacia una posición con un niño este lo recoge automáticamente.
- $sameSpotDirty$: Estoy en la misma posición que una suciedad.
- $sameSpotCorral$: Estoy en la misma posición que un corral.
- $edgeCorral$: Devuelve si la posición actual es de las posiciones vacía del corral más cercana al borde del ambiente.
- $dirtyPerCent$: Detecta el por ciento de suciedad que existe en el tablero.

- $closestPosChild$: Es la posición a la que se debe mover el robot del camino más corto para alcanzar un niño.
- $closestPosDirty$: Es la posición a la que se debe mover el robot del camino más corto para alcanzar una suciedad.
- $closestPosCorral$: Es la posición a la que se debe mover el robot del camino más corto para alcanzar un corral.
- $availablesPositionsToMove$: Es el conjunto de posiciones a las que se puede mover el robot (en caso de no existir ninguna es la propia posición donde se encuentra el robot).

Acciones:

- $robotAction\_cleanDirty$: Limpia la suciedad de la casilla donde se encuentra el robot. Esta acción mantiene al robot en el estado en el que se encuentra. Solo se puede realizar si el robot se encuentra en Estado 1 (Libre)
- $moveRobot\ to\ (x, y)$: Mueve el robot a la casilla (x, y) adyacente que se le indica. Esta acción mantiene al robot en el estado en el que se encuentra. Solo se puede realizar si se encuentra en Estado 1.
- $robotAction\_dropChild$: Suelta al niño que está cargando en este momento el robot. Esta acción cambia el estado del robot a Estado 1. Solo se puede realizar si se encuentra en Estado 2 (Cargando al niño)
- $moveRobotWithChild\ to\ (x, y)$: Mueve al robot mientras carga al niño a la posición (x, y) . Esta acción mantiene al robot en el estado en el que se encuentra. Solo se puede realizar si se encuentra en Estado 2. Como puede moverse 2 pasos la posición (x, y) será cualquier casilla del ambiente a 1 o 2 pasos de distancia del robot.
- $moveRobotRandomDir$: Mueve al robot (y al niño si tiene uno en sus manos) a una posición aleatoria de las posiciones disponibles a las que se puede mover el robot. Esta acción mantiene al robot en el estado en el que se encuentra. Se puede realizar en cualquiera de los 2 estados.

Descripción de agente y su Conducta

1er Agente - Robot Niñera(Proactivo-Reactivo):

La idea de este agente es priorizar recoger a los niños del ambiente y llevarlos hasta el corral. Una vez terminado este trabajo se dedica a limpiar la suciedad. Si no puede alcanzar a ningún niño se dedicará a limpiar la suciedad hasta que pueda encontrar el camino hacia alguno. Si no puede realizar ninguna acción se moverá de forma aleatoria entre las posiciones vacías a que pueda alcanzar.

Conductas:

Las conductas las representaremos de la forma (i, j), donde i es el número representativo del estado del agente (1 si está libre y 2 si carga un niño,) y j un identificador de la j-ésima conducta en ese estado. Por tanto, para realizar la conducta $(1, j)$ se debe cumplir que el robot se encuentra en estado 1

- (1.1) **if** $availableChild$ **then** $moveRobot$ **to** $closestPosChild$
- (1.2) **if** $sameSpotDirty$ **then** $robotAction\_cleanDirty$
- (1.3) **if** $availableDirty$ **then** $moveRobot$ **to** $closestPosDirty$
- (1.4) **if** $True$ **then** $moveRobotRandomDir$ **to** $availablesPositionsToMove$

- (2.1) **if** $edgeCorral$ **and** $sameSpotCorral$ **then** $robotAction\_dropChild$

- (2.2) **if** $sameSpotCorral$ **then** $moveRobotWithChild$ **to** $closestPosEdge$
- (2.3) **if** $availableCorral$ **then** $moveRobotWithChild$ **to** $closestPosCorral$
- (2.4) **if** $True$ **then** $moveRobotRandomDir$ **to** $availablesPositionsToMove$

Relación de orden estricto:

$(1.1) < (1.2) < (1.3) < (1.4)$

$(2.1) < (2.2) < (2.3) < (2.4)$

2do Agente - Robot Aspiradora(Proactivo-Reactivo):

La idea de este agente es priorizar limpiar el tablero por encima de recoger los niños del ambiente. Por tanto el agente buscará si existe alguna suciedad en el ambiente y en caso de ser posible alcanzarla, se moverá en dirección a limpiar esa suciedad. Cuando no existan más suciedades o no tenga ninguna suciedad alcanzable, se dedicará a recoger a los niños. Como el robot no puede limpiar mientras carga a los niños, si este trae a uno consigo y ve una suciedad soltará al niño e irá a buscar la suciedad

Conductas:

- (1.2) **if** $sameSpotDirty$ **then** $robotAction\_cleanDirty$
- (1.3) **if** $availableDirty$ **then** $moveRobot$ **to** $closestPosDirty$
- (1.1) **if** $availableChild$ **then** $moveRobot$ **to** $closestPosChild$
- (1.4) **if** $True$ **then** $moveRobotRandomDir$ **to** $availablesPositionsToMove$

- (2.5) **if** $availableDirty$ **then** $robotAction\_dropChild$
- (2.1) **if** $edgeCorral$ **and** $sameSpotCorral$ **then** $robotAction\_dropChild$
- (2.2) **if** $sameSpotCorral$ **then** $moveRobotWithChild$ **to** $closestPosEdge$
- (2.3) **if** $availableCorral$ **then** $moveRobotWithChild$ **to** $closestPosCorral$
- (2.4) **if** $True$ **then** $moveRobotRandomDir$ **to** $availablesPositionsToMove$

Relación de orden estricto:

$(1.2) < (1.3) < (1.1) < (1.4)$

$(2.5) < (2.1) < (2.2) < (2.3) < (2.4)$

3er Agente - Robot Mixto(Proactivo-Reactivo):

La idea de este agente es realizar acciones mixtas entre el robot aspiradora y el robot niñera basado en el % de suciedad del tablero. Para ello se traza como objetivo cargar niños y llevarlos al corral mientras el % de limpieza este por debajo del 25%, si sube de esta cantidad quiere decir que sus objetivos peligran y puede fallar en cualquier momento, así que pasa a comportarse como el robot aspiradora en ese momento. Para simplificar la escritura de las conductas usaremos

$robotNiñera$ para especificar que realizará la acción correspondiente al robot niñera según las percepciones del ambiente y $robotAspiradora$ para especificar que realizará la acción correspondiente al robot aspiradora según las percepciones del ambiente. En el siguiente ejemplo se mostrará como fuese la verdadera notación de (1.1):

(1.1) **if** $dirtyPerCent < 25$ **and** $availableChild$ **then** $moveRobot$ **to** $closestPosChild$

Conductas:

- (1.5) **if** $dirtyPerCent < 25$ **then** $robotNiñera$
- (2.5) **if** $dirtyPerCent < 25$ **then** $robotNiñera$
- (1.6) **if** $dirtyPerCent \geq 25$ **then** $robotAspiradora$
- (2.6) **if** $dirtyPerCent \geq 25$ **then** $robotAspiradora$

La relación de orden no importaría pues los casos son excluyentes

4to Agente - Robot Aleatorio(Reactivo):

Este agente es uno de pruebas para analizar un comportamiento más tonto respecto al resto. La idea es que se moverá de forma aleatoria entre las posiciones a las que tiene acceso y limpiará si de casualidad se para sobre una casilla sucia. Nunca cargará niños, solo se dedicará a limpiar.

Conductas:

- (1.1) **if** $sameSpotDirty$ **then** $robotAction\_cleanDirty$
- (1.4) **if** $True$ **then** $moveRobotRandomDir$ **to** $availablesPositionsToMove$

Relación de orden estricto

$(1.1) < (1.4)$

Caso Especial - Distribución de trabajos:

Este es un caso especial para probar una mezcla de agentes para ver como se comportarían las simulaciones. Para ello se toma el listado de agentes que se pasan y a la mitad se le otorga la tarea de comportarse como niñeras, mientras que a la otra mitad se le otorga la tarea de comportarse como Aspiradora. De esta manera los robots tendrán tareas secundarias distintas.

Ideas seguidas para la implementación

Para realizar las simulaciones se tomaron en cuenta dos maneras de proceder. Una detallada llamada $SingleSimulation$ y una a gran escala llamada $MultipleSimulation$

Inicialmente se piden los datos al usuario según el tipo de simulación que se desea realizar.

La $SingleSimulation$ se puede utilizar para ver paso a paso como se comporta cada agente y los cambios que se realizan en el ambiente . En este modo se proporciona en consola una representación de como queda el tablero en cada paso. Una vez se muestra el estado actual del ambiente se tienen dos opciones, Si se inserta 1 (se escribe 1 en consola y se pulsa Enter) se efectuará el siguiente paso en la simulación (mover el robot o actualizar el ambiente según corresponda y se mostrará en pantalla). Se se inserta 2 (se escribe 2 en consola y se pulsa Enter) se detendrá automáticamente la simulación y se mostrara en pantalla el % de suciedad que

existía al terminar. Si una simulación llega a un estado terminal (se alcanza la condición de victoria, de derrota) también se detiene automáticamente aunque se inserte 1.

La $MultipleSimulation$ se puede utilizar para correr varias simulaciones y mostrar los resultados finales. Dado los datos de entrada se genera un ambiente de forma aleatoria con esos datos y se procede a simular que sucede con dicho ambiente. Al finalizar se de un resultado de 100 si resultó victoriosa la simulación (Se alcanzó una condición de victoria) y de 0 si resultó en fallida (Se alcanzó la condición de derrota). Luego se procede a generar otro ambiente de forma aleatoria con los datos de entrada y volver a probar hasta realizar $test\_amount$ simulaciones. Luego, se muestra un listado con todos los resultados de las distintas simulaciones y se calcula el % de victoria del agente seleccionado en esas condiciones.

Luego se pide que inserte los datos para rellenar los ambientes de forma aleatoria:

- Largo del ambiente
- Ancho del ambiente
- Cantidad de niños
- Cantidad de obstaculos
- Cantidad de suciedad
- Cantidad de robots
- Tiempo en que demora en cambiar el ambiente (t)
- Numero según tipo de robot que desea utilizar
    - 1 para **Niñera** (Todos los robots tipo Niñera)
    - 2 para **Aspiradora** (Todos los robots tipo Aspiradora)
    - 3 para **Mixto** (Todos los robots tipo Mixto)
    - 4 para **Aleatorio** (Todos los robots tipo Aleatorio)
    - 5 para **Caso Especial** (Mitad de robots son Aspiradora y la otra mitad Niñera)
- Cantidad de Simulaciones que voy a realizar (Solo funciona si es simulación Múltiple)

Explicación de los distintos módulos

modulo Enviroment

En este modulo se encuentra definido el Ambiente y las funciones que modifican a este. Como es un modulo bastante largo se explicarán las más importantes(El resto de funciones tienen comentarios de la tarea que realizan)

La $data\ Enviroment$ es utilizada para guardar todas las $Units$ que conforman el ambiente así como algunos atributos especiales como el ancho y el largo de este.

- $nSize$ es un entero que representa el valor del largo del ambiente
- $mSize$ es un entero que representa el valor del ancho del ambiente
- $emptyList\_env$ es es un listado de Unidades que representa a las casillas vacías del ambiente
- $obstacleList\_env$ es un listado de Unidades que representa a los obstáculos del ambiente
- $dirtyList\_env$ es un listado de Unidades que representa a las casillas sucias del ambiente
- $childList\_env$ es un listado de Unidades que representa a los niños del ambiente
- $robotList\_env$ es un listado de Unidades que representa a los robots del ambiente
- $corralList_env$ es un listado de Unidades que representa a las casillas que son un corral

La $data\ InitialData$ se utiliza para poder acceder a los parámetros de entrada que se dan al programa para realizar las simulaciones. De esta forma puedo volver a generar un ambiente inicial después de haber modificado el actual.

- $nCount$ es un entero que representa el tamaño del largo del ambiente dado por el usuario
- $mCount$ es un entero que representa el tamaño del ancho del ambiente dado por el usuario
- $childCount$ es un entero que representa la cantidad de niños que debe poseer el tablero inicial
- $obstacleCount$ es un entero que representa la cantidad de obstáculos que debe poseer el tablero inicial
- $dirtyCount$ es un entero que representa la cantidad de casillas sucias que debe poseer el tablero inicial
- $robotCount$ es un entero que representa la cantidad de robots que debe poseer el tablero inicial
- $robotTypeIn$ es un entero que representa el tipo de los agentes que se utilizarán

```
data Enviroment = Enviroment {
  nSize::Int,
  mSize::Int,
  boardList_env::[(Int,Int)], --Deprecated
  emptyList_env::[Unit],
  obstacleList_env::[Unit],
  dirtyList_env::[Unit],
  childList_env::[Unit],
  robotList_env::[Unit],
  corralList_env::[Unit]
}deriving Show

data InitialData = InitialData {
  nCount::Int,
  mCount::Int,
  childCount::Int,
  obstacleCount::Int,
  dirtyCount::Int,
  robotCount::Int,
  robotTypeIn::Int
}deriving Show
```

Luego se definen dos funciones para cuando se necesite trabajar con una dirección específica, sobre cada una está comentada lo que significa el número de la dirección

```
--Direcciones alrededro de una casilla, cada direccion se puede referir a la
posicion del numpad respecto el numero 5 como centro
arround_dir::Int->(Int,Int)
arround_dir i
  |i== 1 = ((-1),1)
  |i== 2 = (0,1)
  |i== 3 = (1,1)
  |i== 4 = ((-1),0)
  |i== 5 = (0,0)
  |i== 6 = (1,0)
  |i== 7 = (-1,(-1))
  |i== 8 = (0,(-1))
```

```
  |i== 9 = (1,(-1))

--direcciones
-- 0 = arriba
-- 1 = derecha
-- 2 = abajo
-- 3 = izquierda
dir_x::Int -> Int
dir_x dir
  |dir == 0 = -1
  |dir == 1 = 0
  |dir == 2 = 1
  |dir == 3 = 0
dir_y::Int-> Int
dir_y dir
  |dir == 0 = 0
  |dir == 1 = 1
  |dir == 2 = 0
  |dir == 3 = -1
```

La función $fillUnitType$ y $fillUnitByPairList$ se utilizarán para generar el ambiente inicial, la primera selecciona una posicion aleatoria de la lista de posiciones vacias del enviroment y luego llama a la segunda para que rellene esas posiciones.

```
--Dado un tipo, una cantidad y un listado de casillas vacias crea un listado de
Unidades de tipo type_name de forma aleatoria en esas posiciones
fillUnitType:: StdGen->String->Int->[(Int,Int)] -> [Unit]
fillUnitType g type_name amount emptyList = fillUnitByPairList cellsToFill
type_name
  where cellsToFill = selectRandomFromList g amount emptyList

--Dado un listado de posiciones Crea un listado de unidades de tipo type_name en
esas posiciones
fillUnitByPairList::[(Int,Int)] -> String -> [Unit]
fillUnitByPairList [] _ = []
fillUnitByPairList (x:xs) type_name = (Unit (fst x) (snd x) type_name):
(fillUnitByPairList xs type_name)
```

La función $canMoveChild$ revisa si un niño se puede mover a una dirección especificada. O sea que se cumpla que no se encuentra en un corral , que el niño no esté siendo cargado por un robot, que la casilla adyacente en la dirección seleccionada esté vacía o que tenga un obstáculo el cual se pueda empujar. Si no se cumple ninguna de estas condiciones, no podrá moverse

```
--Devuelve True si el niño puede moverse en una dirección especifica y un Int que
representa la razón de porque se pudo mover
canMoveChild::Enviroment-> Int->Unit->(Bool,Int)
canMoveChild env dir current_child
  |elem (x,y) (getPosList corralL) = (False,0)
  |elem (x,y) (getPosList robotL) && (robotState == "2") = (False,0)
  |elem posToMoveChild (getPosList emptyL) = (True,1)
  |elem posToMoveChild (getPosList obstacleL) = if canMoveObstacles env dir
current_child then (True,2) else (False,0)
```

```
  |otherwise = (False,0)
  where x = xPos current_child
     y = yPos current_child
     xdir = dir_x dir
     ydir = dir_y dir
     posToMoveChild = ((x+xdir),(y+ydir))
     emptyL = emptyList_env env
     obstacleL = obstacleList_env env
     corralL = corralList_env env
     robotL = robotList_env env
     current_robot = getRobotFromList robotL (x,y)
     robotState = getRobotState current_robot
```

Esta función se para saber si un niño puede mover un determinado obstáculo. Para ello se busca el ultimo obstáculo consecutivo dada una posición y la dirección a la que se quiere mover este. Posteriormente se verifica cual es el siguiente elemento en esa misma dirección del ultimo obstáculo. Si ese elemento es una casilla vacía implica que el niño puede mover todos los obstáculos en esa dirección, en caso contrario, no se pueden empujar los obstáculos y se devuelve falso.

```
-- Devuelve True si el obstaculo se puede mover por un niño
canMoveObstacles::Enviroment -> Int -> Unit -> Bool
canMoveObstacles env dir (Unit x y _) =
  let
    lastObst = lastObstacle (obstacleList_env env) x y dir
    current_x = xPos lastObst
    current_y = yPos lastObst
    new_x = current_x + (dir_x dir)
    new_y = current_y + (dir_y dir)
  in elem (Unit new_x new_y "-") (emptyList_env env)

lastObstacle::[Unit]->Int->Int->Int->Unit
lastObstacle obstacleList x_current y_current dir
  |elem new_location (getPosList obstacleList) = lastObstacle obstacleList new_x
new_y dir
  |otherwise = (Unit (x_current) (y_current) "Obstaculo")
  where xdir = dir_x dir
     ydir = dir_y dir
     new_x = (x_current+xdir)
     new_y = (y_current+ydir)
     new_location = (new_x,new_y)
```

Para pintar el enviroment se usa la siguiente estrategia:

Se realiza un listado con todas las listas de $Unit$ que representan casillas del **Ambiente**. Se ordena ese listado para que me queden todas las x de forma consecutiva y creciente seguidas de las y. Luego transformo ese listado con todas las $Units$ en una matriz de $Units$ donde x me representa las **filas** y y las **columnas**. Esta matriz de $Units$ la transformo en una matriz de $Strings$ de forma tal que si dos $Units$ tienen el mismo valor (x, y) entonces pertenecen a la misma $String$. Para terminar tomo esta matriz de $String$ y le realizo un fix para que se vea de forma ordenada cuando la pinte en consola, agregando también una columna y una fila para representar bien las coordenadas de cada elemento.

```haskell
--Transforma el Enviroment en una table de listas de String que se utiliza para pintarla en consola
pipeEnviroment::Enviroment->[[String]]
pipeEnviroment env =
  let
    nLen = nSize env
    mLen = mSize env
    unitList = quicksort(joinEnviromentList env)
    packedList = packEnviroment unitList nLen 0
    packedStrings = packUnitToPackString packedList
    tableString = fixStringOutput packedStrings nLen mLen
  in tableString

  --Empaqueta todos los listados del Enviroment
joinEnviromentList:: Enviroment -> [Unit]
joinEnviromentList (Enviroment _ _ _ e o d c r corral) = e++o++d++c++r++corral

--Empaqueta todos los listados del Enviroment menos empty
joinEnviromentFull:: Enviroment -> [Unit]
joinEnviromentFull (Enviroment _ _ _ _ o d c r corral) = o++d++c++r++corral

--Toma la 1ra letra de un string (como string)
symbolOf::String->String
symbolOf (x:xs) = show x

--Convierte el listado plano de objetos de Enviroment y lo transforma en un listado de listas de objetos del enviroment
--De esta forma tenemos una representacion matricial del enviroment
packEnviroment::[Unit]-> Int -> Int ->[[Unit]]
packEnviroment unitList n i
  |i == n = []
  |otherwise = let
      headList = filter (\u->xPos u == i) unitList
      restList = drop (length headList) unitList
    in headList:packEnviroment restList n (i+1)

--Convierte la matriz de Unidades del Enviroment en una matriz de Strings
packUnitToPackString::[[Unit]]->[[String]]
packUnitToPackString [] = []
packUnitToPackString (x:xs) = let
  headStr = turnUnitListString x 0
  in headStr:packUnitToPackString xs

--Convierte un listado de Unidades en un listado de String, si dos unidades tienen la misma posicion pertenecen al mismo string
turnUnitListString::[Unit]->Int ->[String]
turnUnitListString [] _ = []
turnUnitListString unit@(u:us) i
  |i /= yPos u = turnUnitListString us i
  |otherwise =
  let
    indexPos = yPos u
    unitSamePos = (filter (\x -> yPos x == indexPos) unit)
    slotStr = [head (unit_type x)| x<-unitSamePos]
    tail = drop (length slotStr) unit
  in slotStr:turnUnitListString us (i+1)
```

```
--Arregla el string de salida para que se muestre correctamente en consola y
agrega una fila cabecera que representa el numero de la columna
fixStringOutput::[[String]]->Int->Int->[[String]]
fixStringOutput inString n m =
  let
    columns = map (show) (generateList_0_to_N m)
    fixedHead = map fixChar ("-":columns)
  in fixedHead:(fixStringColumn inString 0)

--Agrega una columna al inicio de cada elemento de la matriz de Unidades de
Enviroment que representa la columna
fixStringColumn::[[String]]->Int->[[String]]
fixStringColumn [] _ = []
fixStringColumn (x:xs) i = let
  fixedHead = map fixChar((show i):x)
  in fixedHead:fixStringColumn xs (i+1)

--Arregla el string de entrada para que coincida con la salida de la consola
fixChar::String->String
fixChar s
  |length s == 0 = " - "
  |length s == 1 = " "++s++" "
  |length s == 2 = s ++ " "
  |otherwise = s
```

--Realiza los cambios en el ambiente dada t unidades de tiempo

El método $updateEnviroment$ es el encargado de modificar el ambiente cada t unidades de tiempo dadas en la entrada del programa. Este método se llamará recursivamente con cada niño del ambiente, los moverá de ser posible a la dirección asignada de forma aleatoria y ensuciará las casillas si estos lograron moverse. Al terminar devolverá un ambiente con todas las actualizaciones realizadas del resultado de mover todos los niños.

Cabe destacar la forma en que se decide que casillas serán las que se ensucien a partir del niño que se logró mover y la cantidad de niños en la cuadrícula de $3x3$. Para ellos se toma de forma aleatoria una de las 8 casillas que rodean al niño actual o la propia casilla que contiene al niño. De esta forma tenemos para elegir entre 9 casillas. Con un random sleccionamos una de forma aleatoria y esa casilla será el centro del $grid$ de $3x3$ del cual contaremos la cantidad de niños para ensuciar dicho $grid$. Nótese que sin importar cual de los 9 $grid$ de $3x3$ que seleccione todos contendrán la posición inicial del niño que se movió. En caso de estar en una esquina del ambiente si se selecciona un centro cuyo alrededor sobresale del ambiente, entonces no se tomará en cuenta esas casillas que están fuera(ya que no se podrán ensuciar ni contendrán niños). Un par de ejemplos a continuación para ejemplificar.

En este caso un niño se encuentra en la casilla (2,4), las casillas azules representan los posibles centros que puedo seleccionar de $grids$ de 3x3.

-	0	1	2	3	4	5	6	7	8
0									
1									
2					N				
3									
4									
5									
6									

Selecciono uno de los posibles centros de forma aleatoria(el que esta coloreado de verde oscuro en la (3,5)) y tomo el $grid$ de $3x3$ que utiliza a esa celda como centro. Luego coloreo de verde claro las celdas alrededor de ese centro para formar el $grid$.

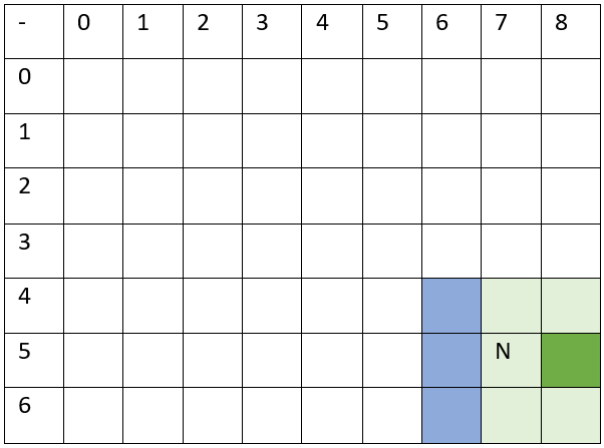
-	0	1	2	3	4	5	6	7	8
0									
1									
2					N				
3									
4									
5									
6									

Cuento la cantidad de niños que hay en las casillas verdes y según esa cantidad genero la suciedad en esas casillas.

En un caso como el siguiente:

-	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									
5								N	
6									

Se usará el $grid 3x3$ que tiene como centro a la casilla de centro verde, pero de ese $grid$ no se tomarán en cuenta las casillas que estén fuera del ambiente(ya que no se podrá generar suciedad ni poseerá niños para contar). Quedando como se muestra en la siguiente figura.



El código para actualizar el ambiente es el siguiente

```
-- recibe un generador, un ambiente ambiente y devuelve como queda el ambiente
después de realizar el cambio al mover todos los niños

updateEnviroment :: StdGen -> Enviroment -> Int -> Enviroment
updateEnviroment g env i = new_env
  where
    new_env = if (i >= length (childList_env env))
    then
      env
    else
      let
        childL = childList_env env
        obstacleL = obstacleList_env env
        dirtL = dirtyList_env env
        emptyL = emptyList_env env
        current_child = (childL)!!i
        i' = i+1
        (dir,newGen) = getDirR g
        xdir = dir_x dir
        ydir = dir_y dir
        current_x = xPos current_child
        current_y = yPos current_child
        new_x = current_x + xdir
        new_y = current_y + ydir
        n = nSize env
        m = mSize env

        (garb',g') = randomR (0,8::Int) g

        (canMakeMove,typeMove) = canMoveChild env dir current_child
        (newChildList1,newObstacleList1,newEmptyList1) = if typeMove == 1 then
moveChild env current_child (new_x,new_y) else (childL,obstacleL,emptyL)
        (newChildList2,newObstacleList2,newEmptyList2) = if typeMove == 2  then
moveChildAndObstacles env current_child dir else (childL,obstacleL,emptyL)


        env'
```

```
          |typeMove == 1 = Enviroment n m (boardList_env env) newEmptyList1
newObstacleList1 (dirtyList_env env) newChildList1 (robotList_env env)
(corralList_env env)
          |typeMove == 2 = Enviroment n m (boardList_env env) newEmptyList2
newObstacleList2 (dirtyList_env env) newChildList2 (robotList_env env)
(corralList_env env)
          |otherwise = env

        selectedGrid = selectRandomGrid g' env' current_child
        (gar,g'') = randomR (0,9::Int) g'
        env''  = generateDirtyFromGrid g'' env' selectedGrid
        env'''
          |typeMove == 1 = env''
          |typeMove == 2 = env''
          |otherwise = env
      in updateEnviroment g'' env''' (i+1)
```

Los siguientes métodos se utilizan para mover niños a casillas vacías, mover niños a una casilla donde existía un obstáculo con anterioridad y como mover un robot. Las 3 funciones son similares, remueven la unidad que se mueve del listado correspondiente y la insertan después en la nueva posición. Se asegura en todo momento de mantener la consistencia del tablero, esto quiere decir que si un niño se mueve deberá insertar una nueva casilla vacía en el listado de casillas vacías si en su posición anterior no existía nada. Lo mismo sucede cuando el robot se mueve. En el caso de un niño moviendo un obstáculo la casilla vacía que se elimina al realizar el movimiento es la que se encuentra una casilla inmediata a partir del ultimo obstáculo encontrada desde el niño respecto a la dirección en la que este se movió.

```
--Mueve el niño a la casilla vacia seleccionada y devuelve un enviroment con las
3 listas que se devuelven en moveChild modificadas
moveChildEnv::Enviroment->Unit->(Int,Int)->Enviroment
moveChildEnv env@(Enviroment n m board e o d c r corral) current_child@(Unit x y
_) (new_x,new_y) =
  let
    (newChildList,obstacleList,newEmptyList) = moveChild env current_child
(new_x,new_y)
  in Enviroment n m board newEmptyList o d newChildList r corral



--Dado un Enviroment, una unidad que deseo mover y una posicion del tablero,
mueve dicha unidad a esa posicion y devuelve 3 listas:
--child,obstacle,empty, con los cambios realizados sobre estas al mover al niño
moveChild::Enviroment->Unit->(Int,Int)->([Unit],[Unit],[Unit])
moveChild env@(Enviroment n m board e o d c r corral) current_child
(new_x,new_y) =
  let
    childList = childList_env env
    emptyList = emptyList_env env
    obstacleList = obstacleList_env env
    current_x = xPos current_child
    current_y = yPos current_child
    new_child = (Unit new_x new_y "Niño")
    new_empty = (Unit current_x current_y "-")
```

```
    oldEnviromentBoard = joinEnviromentFull env
    noEmpty = length ( filter (\x->x==(current_x, current_y)) (getPosList
oldEnviromentBoard) ) /= 1
    current_empty = (Unit new_x new_y "-")
    newChildList = new_child:(remove current_child childList)
    newEmptyList' = remove current_empty emptyList
    newEmptyList = if noEmpty  then newEmptyList' else newEmptyList'++
[new_empty]

  in (newChildList,obstacleList,newEmptyList)

--Mueve el niño a la casilla con obstaculo seleccionada y devuelve como quedan 3
listas del enviroment modificadas
--A diferencia del metodo de arriba este se llama cuando el niño se desea mover a
la direccion de un obstaculo y se da la entrada como una direccion
--En vez de como una posicion
moveChildAndObstacles::Enviroment->Unit->Int->([Unit],[Unit],[Unit])
moveChildAndObstacles env@(Enviroment n m board e o d c r corral) current_child
dir =
  let
    childList = childList_env env
    emptyList = emptyList_env env
    obstacleList = obstacleList_env env
    current_x = xPos current_child
    current_y = yPos current_child
    new_x = current_x + dir_x dir
    new_y = current_y + dir_y dir

    current_obstacle = (Unit new_x new_y "Obstaculo")
    lastObst = lastObstacle obstacleList current_x current_y dir
    new_obstacle_x = (xPos lastObst) + (dir_x dir)
    new_obstacle_y = (yPos lastObst) + (dir_y dir)
    current_empty = (Unit new_obstacle_x new_obstacle_y "-")

    oldEnviromentBoard = joinEnviromentFull env

    noEmpty = length ( filter (\x->x==(current_x, current_y)) (getPosList
oldEnviromentBoard) ) /= 1
    newObstacle = (Unit new_obstacle_x new_obstacle_y "Obstaculo")
    newChild = (Unit new_x new_y "Niño")
    new_empty = (Unit current_x current_y "-")
    newObstacleList = newObstacle:(remove current_obstacle obstacleList)
    newChildList = newChild:(remove current_child childList)
    newEmptyList' = remove current_empty emptyList
    newEmptyList = if noEmpty  then newEmptyList' else newEmptyList'++
[new_empty]

  in (newChildList,newObstacleList,newEmptyList)

--Mueve una unidad a la posicion que se solicita y mantiene la consistencia del
enviroment al realizarlo
moveRobot::Enviroment->Unit->(Int,Int)->Enviroment
moveRobot env@(Enviroment n m board e o d c r corral) u@(Unit x_current
y_current unit_name) (x_new, y_new) =
  let
    robotList = robotList_env env
    childList = childList_env env
```

```
    robotType = getRobotType u
    robotState = getRobotState u
    robot_nameWithChild = "Robot"++robotType++"2"
    new_robotList = if elem (x_new,y_new) (getPosList childList) then (Unit
x_new y_new robot_nameWithChild):(remove u robotList)  else (Unit x_new y_new
unit_name):(remove u robotList)

    new_emptyList' = remove (Unit x_new y_new "-") (emptyList_env env)
    new_empty = (Unit x_current y_current "-")

    oldEnviromentBoard = joinEnviromentFull (Enviroment n m board e d o c
new_robotList corral)
    noEmpty = elem (x_current, y_current) (getPosList oldEnviromentBoard)
    newEmptyList = if noEmpty  then new_emptyList' else new_emptyList'++
[new_empty]

  in Enviroment (nSize env) (mSize env) (boardList_env env) newEmptyList
(obstacleList_env env) (dirtyList_env env) (childList_env env) new_robotList
(corralList_env env)
```

Funciona igual a las funciones mencionadas anteriormente, se encarga de generar suciedad para el ambiente a partir de una casilla 3x3 que se le pasa y la cantidad de niños que se encuentran en estas casillas. Manteniendo la consistencia de las casillas vacías.

```
--Dado un generador , un enviroment y un grid de 3x3, rellena ese grid con
casillas sucias segun la cantidad de niños y actualiza el ambiente
generateDirtyFromGrid::StdGen->Enviroment->[(Int,Int)]->Enviroment
generateDirtyFromGrid g env@(Enviroment n m board e o d c r corral) gridPos =
  let
    emptyList = emptyList_env env
    childList = childList_env env
    dirtyList = dirtyList_env env
    emptyGridPos = (filter (\x -> elem x (getPosList emptyList) ) gridPos)
    childGridPos = (filter (\x -> elem x (getPosList childList) ) gridPos)
    maxDirty = min(length emptyGridPos) (amountOfDirtyByChild (length
childGridPos))
    amountOfDirtyGenerated = randomNumb g 0 (maxDirty)
    (gar,g') = randomR (0,10::Int) g

    fillWithDirtyEmpty = if (length emptyGridPos) == 0  then [] else
selectRandomFromList g' amountOfDirtyGenerated emptyGridPos

    newEmptyList = removePosListFromUnit emptyList fillWithDirtyEmpty
    newDirtyList' = map (generateUnitByPos "Suciedad") (fillWithDirtyEmpty)
    newDirtyList = newDirtyList' ++ dirtyList
  in Enviroment  n m board newEmptyList o newDirtyList childList r corral

--Selecciona la cantidad de suciedad que se va a generar a partir de la cantidad
de niños en la grid

amountOfDirtyByChild::Int->Int
amountOfDirtyByChild child_amount
  |child_amount == 1 = 1
  |child_amount == 2 = 3
  |otherwise = 6
```

Con esta función se genera un nuevo ambiente a partir de una $InitialData$, esta función se llamará en cuando se termine de realizar una simulación y se quiera generar otra con un nuevo ambiente pero los mismos datos iniciales que se le insertaron.

```
generateEnviroment::StdGen->InitialData ->Enviroment
generateEnviroment g initData = env
  where
    n = nCount initData
    m = mCount initData
    child_count = childCount initData
    obstacle_count = obstacleCount initData
    dirty_count = dirtyCount initData
    robot_count = robotCount initData
    robot_type = robotTypeIn initData
  --creando board
    board = boarGeneration n m
    emptyList = fillUnitByPairList board "-"
  --creando posiciones de Corral
    corral = createCorralDefault child_count n m
    emptyList' = removeUnitListFromUnitList emptyList corral
  --creando posiciones de Niños
    childs = fillUnitType g "Niño" child_count (getPosList emptyList')
    emptyList'' = removeUnitListFromUnitList emptyList' childs
    (_,g') = randomR (0,4::Int) g
  --creando posiciones de Obstaculos
    obstacles = fillUnitType g'  "Obstaculo" obstacle_count (getPosList
emptyList'')
    emptyList''' = removeUnitListFromUnitList emptyList'' obstacles
    (_,g'') = randomR (0,5::Int) g'
  --creando posiciones de Suciedad
    dirty = fillUnitType g'' "Suciedad" dirty_count (getPosList emptyList''')
    emptyList'''' = removeUnitListFromUnitList emptyList''' dirty
    (_,g''') = randomR (0,6::Int) g''
  --creando posiciones de Robots
    robots = fillUnitType g''' "Robot" robot_count (getPosList emptyList'''')
    emptyList''''' = removeUnitListFromUnitList emptyList'''' robots
    robotsFixed = setRobotsOptions robots robot_type

    env = Enviroment{
    nSize = n,
    mSize = m,
    boardList_env = board,
    emptyList_env = emptyList''''',
    childList_env = childs,
    obstacleList_env = obstacles,
    dirtyList_env = dirty,
    robotList_env = robotsFixed,
    corralList_env = corral
    }

fixRobotType:: Unit -> String -> Unit
fixRobotType rob typeStr = (Unit (xPos rob) (yPos rob) ("Robot"++typeStr++"1"))

--1 Todos niñera
--2 Todos Aspiradora
```

```
--3 Todos Mixtos
--4 Todos Random
--5 Mitad y Mitad
setRobotsOptions::[Unit] -> Int -> [Unit]
setRobotsOptions [] _ = []
setRobotsOptions robot_list@(r:rs) typeRobot
  |typeRobot == 2 = (Unit x y "RobotS1"):setRobotsOptions rs typeRobot
  |typeRobot == 3 = (Unit x y "RobotM1"):setRobotsOptions rs typeRobot
  |typeRobot == 4 = (Unit x y "RobotR1"):setRobotsOptions rs typeRobot
  |typeRobot == 5 = distributeRobots robot_list (length robot_list)
  |otherwise = (Unit x y "RobotN1"):setRobotsOptions rs typeRobot
  where
    x = xPos r
    y = yPos r
```

modulo UnitClass

En este módulo tenemos la data $Unit$, que representa una Unidad. Una unidad es una casilla del tablero, puede ser un niño, un robot, una suciedad, obstáculo, corral o casilla vacía. Cada unidad estará compuesta por dos int: $xPos$ y $yPos$ que representan la posición de esa $Unit$ en el ambiente y un String $unit\_type$ que representa el tipo de esa $Unit$ (si son robot, niño, corral ,etc). Los robots tienen un tipo de la forma "**RobotAB**". Donde A es una letra que representa el tipo de robot y "B" un número que representa su estado. Por ejemplo el "**RobotN1**" quiere decir que la unidad es un Robot Niñera que se encuentra en estado 1(Libre), mientras que "RobotS2" quiere decir que la unidad es un Robot Aspiradora que se encuentra en estado 2(Cargando a un niño). La data $Unit$ se puede mostrar en consola y se pueden igualar entre ellas. Se tiene una instancia de **Ord** para definir la desigualdad entre dos $Unit$ que está dada al comparar las x y las y de cada $Unit$. La definición de la data $Unit$ y la forma en que se compra podemos verla a continuación.

```
data Unit = Unit{
          xPos::Int,
          yPos::Int,
          unit_type::String
        } deriving (Show,Eq)

instance Ord Unit where
  (<) (Unit x1 y1 t1) (Unit x2 y2 t2)
    |x1 < x2 = True
    |x1 == x2 = y1 < y2
    |otherwise = False
  (<=)(Unit x1 y1 t1) (Unit x2 y2 t2)
    |x1 < x2 = True
    |x1 == x2 = y1 <= y2
    |otherwise = False
  (>) u1 u2 = not (u1<=u2)
  (>=) u1 u2 = not (u1<u2)
```

Para los $Unit$ de tipo robot se le piden su estado y su tipo con las siguientes funciones

```
--Devuelve el tipo de un robot que se le asigno una tarea
--Types
--"N" == Niñera
--"S" == Aspiradora
--"M" == Mixto
--"R" == aleatorio
getRobotType:: Unit -> String
getRobotType u = [r_type!!5]
    where r_type = unit_type u

--Devuelve el estado de un robot
-- Status
-- 1 Libre
-- 2 Cargando niño
getRobotState:: Unit->String
getRobotState u = [r_status!!6]
  where r_status = unit_type u
```

Con esta función puedo preguntar si un robot en una posición determinada pertenece al listado de Robots. De esta forma no tengo que saber el tipo o estado en el que se encuentra para preguntar

```
--Devuelve si un elemento de una lista de Unidades es un Robot, ignorando tipo y
estado
elemRobot::Unit-> [Unit]->Bool
elemRobot unt (u:us)
  |isRobot = True
  |otherwise = elemRobot unt us
  where isRobot = (take 5 (unit_type unt)) == "Robot"
```

Con esta función $getPos$ dada una $Unidad$ me devuelve un par (x, y) con las coordenadas de esa $Unidad$ en el ambiente y con $getPosList$ puedo hacer lo mismo de un **listado** de $Unidades$

```
--Devuelve la posicion x,y de la unidad en forma de tupla
getPos:: Unit -> (Int,Int)
getPos u = (xPos u,yPos u)

--Dado un lsitado de Unidades devuelve un listado de tuplas con las posiciones de
esas unidades
getPosList::[Unit]->[(Int,Int)]
getPosList [] = []
getPosList (u:us) = (getPos u):getPosList us
```

La función $removeUnitListFromUnitList$ se utiliza para eliminar los elementos de la 2da lista que se encuentran en la 1ra, sirve para actualizar listas con una sola función (Mantener la consistencia de la lista de Unidades que representan casillas vacías en la generación del ambiente o cuando genero suciedad). Similar es la $removePosListFromUnit$ pero en vez de eliminar

$Unidades$ a partir de otro listado de $Unidades$, las elimina a partir de un listado de tuplas de **posiciones**.

```
--Dado dos listas de Unit elimina de la 1ra lista los Unit que tengan la misma
posicion que los que se encuentran en la 2da
removeUnitListFromUnitList::[Unit]->[Unit]->[Unit]
removeUnitListFromUnitList u1 u2 = removePosListFromUnit u1 (getPosList u2)

--Dada una lista de Unidades y una lista de posiciones, remueve todas las
unidades que se encuentren en alguna posicionn de la lista de posiciones

removePosListFromUnit::[Unit]->[(Int,Int)]->[Unit]
removePosListFromUnit list [] = list
removePosListFromUnit [] _ = []
removePosListFromUnit unitA unitB = filter (\x ->not( elem (getPos x) unitB ))
unitA
```

Esta función se usa para a partir de un listado de posiciones y un nombre de tipo poder generar Unidades en esa posición

```
--Dado un nombre de una Unit y una posicion, genera la unidad en esa posicion
generateUnitByPos::String->(Int,Int) ->Unit
generateUnitByPos name (x,y) = (Unit x y name)
```

modulo BfsRobot

En este modulo se encuentra el bfs que se realiza para encontrar otras $Units$ a partir de la posición de un Robot. De esta forma podemos saber si existe un camino hasta esa posición y cual es la casilla a la que debemos movernos para llegar a ese posición. En caso de no encontrar ningún camino se devuelve $(-1, -1)$

La función $findBFS$ realiza un bfs sobre el ambiente a partir de un robot dado. Con la función $AvailablePos$ puedo saber a que casillas se puede mover un robot.

```
--Dado un enviroment, una unidad y en string de busqueda, devuelve la posicion en
la que se encuentra la unidad de tipo: <seachString> más cercana
findBFS::Enviroment-> Unit->String-> [(Int,Int)]
findBFS env current_robot searchString = bfsRobot env [current_robot]
[(current_robot,current_robot)] searchString
--Dado un enviroment, una cola, un listado de casillas visitadas  y el tipo de
unidad que estoy buscando, devuelve el camino que se necesita recorrer
-- para encontrar esa unidad
bfsRobot::Enviroment -> [Unit] -> [(Unit,Unit)]->String -> [(Int,Int)]
bfsRobot env queue visited searchString
  |length queue == 0 = [(-1,-1)]
  |unit_type current_node == searchString = checkEndbfs env current_node visited
((-1),(-1))
  |otherwise =
    let
      new_queue' = tail queue
```

```
      (new_queue,new_visited) = lookAroundAndQueue env current_node new_queue'
visited searchString
    in bfsRobot env new_queue new_visited searchString
  where
    current_node = head queue
--Se llama al encontrar una unidad al final del bfs y recorre el camino a la
inversa para obtener que camino debo recorrer
checkEndbfs:: Enviroment -> Unit -> [(Unit,Unit)]-> (Int,Int) -> [(Int,Int)]
checkEndbfs env current_node visited (x_old,y_old)
  |parent_node == current_node = []
  |otherwise = (checkEndbfs env parent_node visited (x_current,y_current))++
[(x_current,y_current)]
  where
    parent_node = snd (head (filter (\x->fst x == current_node) visited))
    x_current = xPos current_node
    y_current = yPos current_node
--Agrupa todas las posiciones a las que se puede mover una unidad dada en el
enviroment
lookAroundAndQueue::Enviroment -> Unit ->[Unit]-> [(Unit,Unit)]-> String ->
([Unit],[(Unit,Unit)])
lookAroundAndQueue env current_node queue visited searchString=
  let
    x_current = xPos current_node
    y_current = yPos current_node
    dir1 = 0
    dir2 = 1
    dir3 = 2
    dir4 = 3
    childList = childList_env env
    childPosList = getPosList childList
    (available_pos1,new_unit1) = availablePos env current_node dir1 searchString
    -- isChild1 = elem (x_current + dir_x dir1 , y_current + dir_y dir1)
childPosList
    visited1 = elem new_unit1 (map fst visited)
    (available_pos2,new_unit2) = availablePos env current_node dir2 searchString
    -- isChild2 = elem (x_current + dir_x dir2 , y_current + dir_y dir2)
childPosList
    visited2 = elem new_unit2 (map fst visited)
    (available_pos3,new_unit3) = availablePos env current_node dir3 searchString
    -- isChild3 = elem (x_current + dir_x dir3 ,y_current + dir_y dir3)
childPosList
    visited3 = elem new_unit3 (map fst visited)
    (available_pos4,new_unit4) = availablePos env current_node dir4 searchString
    -- isChild4 = elem (x_current + dir_x dir4 , y_current+dir_y dir4)
childPosList
    visited4 = elem new_unit4 (map fst visited)
    toVisit1 = if available_pos1 && not visited1 then [(new_unit1,current_node)]
else []
    toVisit2 = if available_pos2 && not visited2 then [(new_unit2,current_node)]
else []
    toVisit3 = if available_pos3 && not visited3 then [(new_unit3,current_node)]
else []
    toVisit4 = if available_pos4 && not visited4 then [(new_unit4,current_node)]
else []
    toQueue1 = map fst toVisit1
    toQueue2 = map fst toVisit2
    toQueue3 = map fst toVisit3
    toQueue4 = map fst toVisit4
```

```
    new_queue = queue++toQueue1++toQueue2++toQueue3++toQueue4
    new_visit = visited++toVisit1++toVisit2++toVisit3++toVisit4
  in (new_queue,new_visit)



--Posiciones a las que se puede mover un robot:
--No hay robot en esa posicion
--Es una posicion vacia
--Es una posicion sucia
--Es una posicion con corral sin niño
--Es una posicion con niño que no esta en un corral
availablePos:: Enviroment-> Unit-> Int-> String->(Bool,Unit)
availablePos env (Unit x_current y_current _) dir searchString
  |robotSpot = (False,(Unit (-1) (-1) "ERROR"))
  |emptySpot = (True,emptyUnit)
  |dirtySpot = (True, dirtyUnit)
  |corralSpot && not childSpot = (True, corralUnit)
  |childSpot && (searchString == "Niño") && not corralSpot = (True,childUnit)
  |otherwise = (False,(Unit (-1) (-1) "ERROR"))
  where
    emptyList = emptyList_env env
    dirtyList = dirtyList_env env
    corralList = corralList_env env
    childList = childList_env env
    robotList = robotList_env env
    obstacleList = obstacleList_env env
    x_new = x_current + dir_x dir
    y_new = y_current + dir_y dir
    new_location = (x_new,y_new)
    emptyUnit = (Unit x_new y_new "-")
    emptySpot = elem emptyUnit emptyList
    dirtyUnit = (Unit x_new y_new "Suciedad")
    dirtySpot = elem new_location (getPosList dirtyList)
    corralUnit = (Unit x_new y_new "Corral")
    corralSpot = elem new_location (getPosList corralList)
    childUnit = (Unit x_new y_new "Niño")
    childSpot = elem new_location (getPosList childList)
    robotUnit = (Unit x_new y_new "Robot")
    robotSpot = elem new_location (getPosList robotList)
```

modulo RobotBehavior

En este modulo se encuentra el comportamiento de cada uno de los robots según el tipo que se le asigna, un método $updateRobot$, con el cual solo tengo que realizar un llamado a la función con el ambiente correspondiente y esta se encargará de hacer los llamados a las funciones correspondientes que se encargan de actualizar a los robots según su tipo y el estado en el que se encuentran. Todas las funciones están comentadas con su funcionalidad.

```
--Todos los robots del enviroment realizan su accion correspondiente
updateRobot::StdGen -> Enviroment-> Enviroment
updateRobot g env = updateRobot_list g env (robotList_env env)
--Pasa por cada robot del listado de robots del enviroment y va haciendo que
realicen sus movimientos uno a uno
```

```haskell
updateRobot_list::StdGen -> Enviroment -> [Unit] -> Enviroment
updateRobot_list g env [] = env
updateRobot_list g env@(Enviroment n m b e o d c robotList corral) (r:rs) =
  let
    updatedEnvairoment = updateRobot_ByType g env r
    (_,g') = randomR (1,11::Int) g
  in
    updateRobot_list g' updatedEnvairoment rs

--Realiza la accion correspondiente segun el tipo que es el robot
updateRobot_ByType::StdGen -> Enviroment -> Unit -> Enviroment
updateRobot_ByType g env current_robot
  |robot_type == "N" = updateRobotNiño g env current_robot
  |robot_type == "S" = updateRobot_Aspiradora g env current_robot
  |robot_type == "M" = updateRobot_Mixto g env current_robot
  |robot_type == "R" = updateRobot_Random g env current_robot
  |otherwise = []!!10
  where
    robot_type = getRobotType current_robot

--Mueve un robot encargado de transportar niños
updateRobotNiño::StdGen->Enviroment -> Unit -> Enviroment
updateRobotNiño g env current_robot
    --Acciones según la percepcion
    |robotState == "1" && (availableChild /= [((-1),(-1))]) = moveRobot env
current_robot posToMove_RobotToChild
    |robotState == "1" && sameSpotDirty =  robotAction_cleanDirty env
current_robot
    |robotState == "1" && (availableDirty /= [((-1),(-1))]) = moveRobot env
current_robot posToMove_RobotToDirty
    |robotState == "2" && not aboveCorral && sameSpotCorral =
robotAction_dropChild env current_robot
    |robotState == "2" && sameSpotCorral && corralEmpty = moveRobotWithChild env
current_robot ((x_current-1),(y_current))
    |robotState == "2" && (availableCorral /= [((-1),(-1))]) =
moveRobotWithChild env current_robot posToMove_RobotToCorral
    |otherwise = moveRobotRandomDir g env current_robot
    -- |otherwise = env
  where
    --Estado del robot
    -- |1 para libre
    -- |2 para cargando al niño
    robotState = getRobotState current_robot
    x_current = xPos current_robot
    y_current = yPos current_robot

    --Percepciones
    --Si hay al menos un niño alcanzable
    availableChild = findBFS env current_robot "Niño"
    posToMove_RobotToChild = head availableChild
    --Si hay al menos una suciedad alcanzable
    availableDirty = findBFS env current_robot "Suciedad"
    posToMove_RobotToDirty = head availableDirty
    --Si hay al menos un corral alcanzable
    availableCorral = findBFS env current_robot "Corral"
    amountToMove = take 2 availableCorral
    posToMove_RobotToCorral = if length amountToMove == 1 then head
availableCorral else availableCorral!!1
```

```
    --Si estoy sobre un niño
    sameSpotChild = elem (Unit x_current y_current "Niño")  (childList_env env)
    --Si estoy sobre una suciedad
    sameSpotDirty = elem (Unit x_current y_current "Suciedad") (dirtyList_env
env)
    --Si estoy sobre un corral
    sameSpotCorral = elem (Unit x_current y_current "Corral") (corralList_env
env)
    --Si la posicion superior a la mia es un corral
    aboveCorral = elem (Unit (x_current-1) (y_current) "Corral") (corralList_env
env) && not (elem (Unit (x_current-1) (y_current) "Niño") (childList_env env))
    --Si el corral de encima está vacio
    corralEmpty = checkEmptyCorral env (x_current-1,y_current)




--Agente Random
updateRobot_Random::StdGen -> Enviroment -> Unit ->Enviroment
updateRobot_Random g env current_robot@(Unit x y robot_name)
  |sameSpotDirty =  robotAction_cleanDirty env current_robot
  |otherwise = moveRobotRandomDir g env current_robot
  where
    sameSpotDirty = elem (Unit x y "Suciedad") (dirtyList_env env)

--Agente aspiradora
updateRobot_Aspiradora::StdGen -> Enviroment -> Unit ->Enviroment
updateRobot_Aspiradora g env current_robot
    |robotState == "1" && sameSpotDirty =  robotAction_cleanDirty env
current_robot
    |robotState == "1" && (availableDirty /= [((-1),(-1))]) = moveRobot env
current_robot posToMove_RobotToDirty
    |robotState == "1" && (availableChild /= [((-1),(-1))]) = moveRobot env
current_robot posToMove_RobotToChild
    |robotState == "2" && (availableDirty /= [((-1),(-1))]) =
robotAction_dropChild env current_robot
    |robotState == "2" && not aboveCorral && sameSpotCorral =
robotAction_dropChild env current_robot
    |robotState == "2" && sameSpotCorral && corralEmpty = moveRobotWithChild env
current_robot ((x_current-1),(y_current))
    |robotState == "2" && (availableCorral /= [((-1),(-1))]) =
moveRobotWithChild env current_robot posToMove_RobotToCorral
    |otherwise = moveRobotRandomDir g env current_robot
  where
    --Estado del robot
    -- |1 para libre
    -- |2 para cargando al niño
    robotState = getRobotState current_robot
    x_current = xPos current_robot
    y_current = yPos current_robot
    --Percepciones
    --Si hay al menos un niño alcanzable
    availableChild = findBFS env current_robot "Niño"
    posToMove_RobotToChild = head availableChild
    --Si hay al menos una suciedad alcanzable
    availableDirty = findBFS env current_robot "Suciedad"
    posToMove_RobotToDirty = head availableDirty
```

```
    --Si hay al menos un corral alcanzable
    availableCorral = findBFS env current_robot "Corral"
    amountToMove = take 2 availableCorral
    posToMove_RobotToCorral = if length amountToMove == 1 then head
availableCorral else availableCorral!!1
    --Si estoy sobre un niño
    sameSpotChild = elem (Unit x_current y_current "Niño")  (childList_env env)
    --Si estoy sobre una suciedad
    sameSpotDirty = elem (Unit x_current y_current "Suciedad") (dirtyList_env
env)
    --Si estoy sobre un corral
    sameSpotCorral = elem (Unit x_current y_current "Corral") (corralList_env
env)
    --Si la posicion superior a la mia es un corral
    aboveCorral = elem (Unit (x_current-1) (y_current) "Corral") (corralList_env
env) && not (elem (Unit (x_current-1) (y_current) "Niño") (childList_env env))
    --Si el corral de encima está vacio
    corralEmpty = checkEmptyCorral env (x_current-1,y_current)



--Agente mixto
updateRobot_Mixto::StdGen -> Enviroment -> Unit ->Enviroment
updateRobot_Mixto g env current_robot =
    let
      porCientoDeSuciedad = dirtyAmount env > 25
      condicionBuena = updateRobotNiño g env current_robot
      condicionMala = updateRobot_Aspiradora g env current_robot
    in if porCientoDeSuciedad then condicionMala else condicionBuena
```

modulo Utils

En este modulo tenemos las funciones de utilidad que tienen diversos usos. Se mostrarán aquí algunos ejemplos, el resto se pueden ver en $utils.hs$, todas las funciones están comentadas con su funcionalidad.

Para poder generar listas del Ambiente a partir de n y m

```
--Genera una lista con los elementos desde N hasta 0, esto se utiliza después
para generar las casillas del ambiente
generateList_N_to_0:: Int -> [Int]
generateList_N_to_0 0 = []
generateList_N_to_0 n = (n-1):generateList_N_to_0 (n-1)

--Se utiliza principalmente para generar los ambientes
```

```
--Genera una lista con los elementos desde 0 hasta N
generateList_0_to_N::Int -> [Int]
generateList_0_to_N n = reverse (generateList_N_to_0 n)
```

Para poder pintar los ambientes es necesario hacer un ordenamiento de listas ya que durante la adición y substracción de elementos estos se desordenan.

```
--Se utiliza para ordenar listas en caso de que se necesite
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort lesser ++ [x] ++ quicksort greater
    where
        lesser  = filter (< x) xs
        greater = filter (>= x) xs
```

En repetidas ocasiones es necesario borrar elementos de una lista (como por ejemplo a la hora de eliminar la suciedad o de mover un niño de una casilla a otra). Si el elemento de entrada es igual al cabecera de la lista fabrico la lista de llamar recursivamente a la función con el cuerpo, en caso contrario me quedo con la lista que se genera con la cabecera y llamar recursivamente con el cuerpo

```
----Elimina todos los elementos de un listado que sean iguales a un elemento que
se le da de entrada
remove::(Eq a)=> a -> [a] -> [a]
remove _ [] = []
remove u (x:xs)
  | u == x = remove u xs
  | otherwise = x:(remove u xs)
```

modulo RandomUtils

En este modulo tenemos funciones que realizan operaciones de aleatoriedad. Al igual que con el modulo $Utils$ se mostrarán aquí algunos ejemplos, el resto de funciones que se utilizan se pueden ver en $randomUtils.hs$ con un comentario explicando para que se utilizan

Se utilizará esta función para generar el ambiente o seleccionar una dirección a la que mover el robot de forma aleatoria según un listado de movimientos posibles. Cada vez q haga falta seleccionar elementos de formas aleatorias de listas.

La idea es tener una función que dada una lista de entrada devuelva la misma lista per con sus valores en posiciones aleatorias, para ello se selecciona un valor aleatorio entre 0 y el $length$ de la lista de entrada, se borra ese valor de la lista y se llama recursivamente con la lista sin el valor seleccionado y un nuevo generador.

Como solo quiero tomar n elementos de la lista de entrada, hago $take\ n$ de esa función auxiliar

```
selectRandomFromList::(Eq a) => StdGen -> Int -> [a] -> [a]
selectRandomFromList g n list = take n (selectRandomFromListOptimiced g list)

selectRandomFromListOptimiced::(Eq a)=> StdGen -> [a] -> [a]
selectRandomFromListOptimiced _ [] = []
selectRandomFromListOptimiced g listIn =
  let
    len = length listIn
    i = randomNumb g 0 len
    (_,g') = randomR (0,20::Int) g
    selectedElem = listIn!!i
    newList = remove selectedElem listIn
  in (selectedElem:(selectRandomFromListOptimiced g' newList))
```

Consideraciones obtenidas a partir de la ejecución de las simulaciones del problema (determinar a partir de la experimentación cuales fueron los modelos de agentes que mejor funcionaron)

Se probaron distintos ambientes con diferentes datos de entrada, a continuación se presentará una tabla con los distintos casos que se probaron.

Por cada fila se muestran los datos que se utilizaron para generar 100 ambientes distintos en los cuales se realizó una simulación. El % que sale debajo de cada robot representa el % de victoria (O sea la cantidad de veces que se alcanzó la condición de victoria en las simulaciones respecto a la cantidad de simulaciones realizadas con esos datos).

```
"Simulating.... please wait"
"[0,100,100,100,0,0,0,0,0,0,0,0,100,0,0,0,0,0,0,0,0,100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,100,0,0,0,100,0,0,0,0,0]"
"Ended the 50 Simulation: "
"Victory rate is: 14%"
```

Esta salida que se muestra al finalizar todas las simulaciones son los resultados de generar un ambiente y simular el comportamientos de los agentes hasta alcanzar una condición de victoria o de derrota.

En el ejemplo mostrado en imagen se realizaron 50 simulaciones con distintos ambientes cada una, donde la 1ra paró al alcanzar una condición de derrota, la 2da una condición de victoria, la 3ra de victoria y así sucesivamente

Los resultados de las simulaciones realizadas se muestran en la siguiente tabla

Ambiente	Niños	Robots	Obstáculos	Suciedad	t	Niñera	Aspir.	Mixto	Aleatorio	Especial
(20,20)	10	1	10	5	3	92%	24%	78%	8%	86%
(20,20)	10	4	10	5	3	100%	100%	100%	12%	100%
(20,20)	30	10	10	10	4	96%	100%	98%	0%	100%
(10,10)	10	3	5	5	2	52%	0%	22%	2%	42%
(10,10)	5	2	5	0	1	96%	34%	96%	2%	90%
(15,10)	10	5	8	5	2	94%	6%	84%	0%	86%
(15,10)	15	3	0	20	5	98%	100%	100%	0%	100%
(15,15)	20	3	10	20	3	10%	0%	16%	0%	14%

Conclusiones del Robot Niñera:

Es el robot que mostró los mejores resultados para todos los casos de entrada que se le asignaron. Por tanto podemos ver que es el más eficiente del grupo a la hora de cumplir el objetivo principal.

Conclusiones del Robot Aspiradora

Sus resultados son muy inestables, pueden pasar de ser muy eficiente y lograr mantener el ambiente limpio a tener un % de victoria extremadamente bajo en los ambientes más difíciles de mantener limpieza debido a la cantidad de niños y el tamaño de este. Si el robot pudiese limpiar con un niño en mano probablemente fuese más eficiente ya que no tendría que soltar al niño para limpiar la suciedad lo que hace que malgaste un turno

Conclusiones del Robot Mixto

Sus resultados son algo inestables, pero por lo general es casi tan eficiente como el niñera, en algunos casos tiene un % de victoria superior al robot Niñera pero en ambientes desfavorables (los que por norma general todos los agentes tuvieron un % de victoria bajo), sus resultados fueron inferiores al robot Niñera. Por tanto la estrategia de tener un determinado comportamiento según el % de suciedad no mejora la idea principal del Niñera. Si el robot Aspiradora tuviese más facilidades quizás el mixto también mejorase.

Conclusiones del Robot Aleatorio

La estrategia de moverse de forma aleatoria para limpiar el ambiente es extremadamente ineficiente. Es el agente que menos % de victoria tiene de todos por gran diferencia. En ambientes fáciles de resolver para otros agentes, este mantiene un % de victoria bajo y en ambientes desfavorables para mantener la limpieza casi nunca logra alcanzar la condición de victoria. Esto nos da una idea de que tener objetivos secundarios es mucho más eficiente que moverse sin rumbo.

Conclusiones de la División de Tareas

Esta estrategia tiene resultados muy similares al robot niñera. No podemos concluir que se más eficiente que un robot Niñera debido a que la cantidad de simulaciones que se realizaron no es extenuantemente grande. Por lo que no se puede llegar a algo concluyente de que tan eficiente sería respecto al Niñera. Lo que si se puede observar es que resulta ser el que tiene mejores resultados respecto al Mixto (y por consiguiente al aspiradora)

Requisitos Para correr el programa:

- Tener instalado algún compilador de haskell.
- Tener instalada la librería $System.Random$