Project Functioneel Programmeren

Taakverdeling

Maximiliaan Leyman

- Interpreter
- MBot

Benjamin Rombaut

- Simulator

Inleiding

We hebben een simulator gemaakt zoals beschreven in de opdracht. De functionaliteit van lijnen trekken en niet door muren rijden intbreekt wel. Daarbij geeft de Linereader sensorwaarde ook enkel BOTHB of BOTHW terug. De simulator is te besturen met het toetsenbord en te testen met een Main.hs file in /src waarin de gevraagde programma's in Haskell vorm staan. De graphics en grid representatie file hebben wel een absoluut path, dus dit moet handmatig aangepast worden.
De parser…

Syntax van de taal

- Geeft een overzicht van de constructies in je taal in (informele) BNF vorm.

Semantiek van de taal

- Voor elk van je taalconstructies geef je een korte uitleg wat de taalconstructies doen en hoe je deze gebruikt.

Voorbeelden programma's

- Geef volledige uitleg bij de programma's die je geïmplementeerd hebt in je eigen programmeertaal.
- Deze programma's moet je ook als aparte files indienen bij je project.
- Lights
- Follow Line
    - If((Right is black) AND (Left is black))
    - ---- Go forward
    - Else if((Right is white) AND (Left is black))
    - ---- Turn Left
    - Else ((Right is black) AND (Left is white))
    - ---- Turn Right
    - Else ((Right is white) AND (Left is white))
    - ---- Aaargh!! I'm lost AGAIN?!?
- Avoid obstacles

Implementatie

Voor de Simulator is er vertrokken vanuit het uiteindelijke resultaat van de Sokoban oefening. De module exporteert hetzelfde interface als het Mbot package. Er zijn enkele constanten bovenaan gedefinieerd. Dan zijn de types en data. Data Robot bevat de sensordata en zijn positie. Data World bevat de Robot, muren, lijnen, afmetingen van het veld en de muispositie. Er is voor geopteerd om het Device voor te stellen als 3 Mvars, waarmee de sensordata en commando's gecommuniceerd kunnen worden. De constructors van data Command stellen

dan weer de verschillende soorten commando's voor. Deze worden dan ook in hun functievorm geexporteerd als de Mbot functies.
openMBot leest de wereld, maakt de Device en geeft deze door aan een nieuwe thread waarin de Gloss playIO functie wordt uitgevoerd. Dan returned hij de Device zodat het programma dat deze functie aanroept dit kan gebruiken voor de Mbot commando's.

We probeerden zoveel mogelijk de geometrie functies uit Graphics.Gloss.Geometry (normalizeAngle, vectorbewerkingen…) te gebruiken. De positiefuncties voor een diferentiaalmotor werden meegegeven in de opgave. Extra functies waren een getLineSensorPoints voor punten voor de twee sensoren te krijgen net naast de robotpositie, afhankelijk van de hoek theta. Ook een getSightline voor de afstandssensor, eenvoudig in vectorvorm. getSegments alle 4 de muren van een blok. getRectFromSeg was voor een dikkere lijn te tekenen, maar is ongebruikt. Soms kon een functie niet volledig hergebruikt worden voor zowel de render als simulate functie omdat de eerste in pixelcoordinaten werkt en de ander in gridcoordinaten, dus werd er een extra variabele meegegeven die de constante lineThickness of wallSize aanpaste naargelang het nodige.

In makeWorld wordt de zelfde methode als in Sokoban gebruikt om een tekstgrid in te lezen. Wel was er een aanpassing nodig voor de y-as naar onder ipv naar boven. De lijnen werden apart ingelezen, met pattern matching als parser. De afmetingen van de viewport wordt ook berekend.

De rendermethode mapt met verschillende rendermethodes over de posities en afbeeldingen. Er is een renderDebugText zodat ik visuele output is van de gekozen waarden. toPixelCoord gebruikt de constante cell om alles coordinaten uit te vergroten.

handleEvent verzorgt besturing met de pijltoetsen van de Robot. Delete stopt de Robot en de spatiebalk zet de Robot terug op het scherm.

Simulate world verplaatst de Robot, update de sensorwaarden en communiceert met de Device. De afstand tot een muur wordt bepaald door het minimum van de afstand met al de mogelijke snijpunten van de line of sight met de 4 zijden van alle Walls. De LineReader sensor bepaald voor beide sensorpunten of de afstand tot een lijnstuk kleiner is dan de helft van de lijndikte, enkel indien dit lijnstuk snijdt met het lijnstuk van de twee sensorpunten.

executeCommand gebruikt pattern matching om het commande uit te voeren op de World. Hierdoor verkrijgen we een IO World, vandaar dat we geen pure Gloss play functie kunnen gebruiken en het playIO moet zijn.

Conclusie

In de simulator kon de ultra sonic sensor beter geïmplementeerd worden als een arc van 45° in plaats van een lijn, zoals in het live voorbeeld is gezien. De Mbot zou dan minder kans hebben om langs muren te schuren met deze extra data. De waarden doorgegeven aan de simulator of de Mbot zoals snelheid verschillen ook, het zou dus interessant zijn om die voor de simulator te mappen zodat deze hetzelfde kunnen zijn. Zoals in de inleiding aangegeven, ontbreekt de functionaliteit om lijnen te trekken als gebruiker en precieze Line data te verkrijgen uit de gesimuleerde sensor.

Appendix Broncode

De nummering van onderstaande code loopt niet correct door, aangezien sommige lijnen breder zijn dan de paginabreedte.

```
module Simulator (openMBot,
                  closeMBot,
                  readUltraSonic,
                  readLineFollower,
                  sendCommand,
                  playTone,
                  setMotor,
                  setRGB
                 ) where

import Data.List (delete, sort, maximumBy)
import Data.Ord (comparing)
import qualified Graphics.Gloss.Interface.IO.Game as G
import Graphics.Gloss.Data.Bitmap (loadBMP)
import Graphics.Gloss.Data.Color
import Graphics.Gloss.Data.Point
import Graphics.Gloss.Data.Vector
import Graphics.Gloss.Geometry.Line
import Graphics.Gloss.Geometry.Angle
import System.Environment (getArgs)
import Control.Concurrent
import Control.Monad (join)
import Control.Arrow ((***))
import Data.Maybe (catMaybes, mapMaybe)

-- CONSTANTS --
initLED = (255, 255, 255)
initRobot = Robot (0,0) 0 0 radiusWheel distBetweenWheels 0 maxDistSensor BOTHW
(initLED, initLED)
-- In the beginning, the world is empty, except for a robot at (0, 0).
emptyWorld = World initRobot [] [] 0 (0,0)
initWindow = G.InWindow "MBot" (1000,1000) (100,100)

-- Radius of wheel
radiusWheel = 20
-- Distance between wheels
distBetweenWheels = 20
-- Distance between LineFollowing sensors
distLineSensors = 1
-- Maximum detection distance for ultra sonic sensor
maxDistSensor = 3
-- Thickness of lines
lineThickness = 20
defaultSpeed = 0.05

-- Pixel size of a single wall
cell = 32
```

```
48
49 -- TYPES --
50 -- R G B values 0 - 255; (0, 0, 0) is off
51 type LED = (Int, Int, Int)
52 type Seg = (G.Point, G.Point)
53
54 -- DATA --
55 data Robot   = Robot { rCoord :: G.Point,
56                        rVl :: Float, -- Speed of left wheel
57                        rVr :: Float, -- Speed of right wheel
58                        rWr :: Float, -- Radius of wheel
59                        rWa :: Float, -- Distance between wheels
60                        rTheta :: Float, -- Direction in radials
61                        rDistance :: Float, -- Distance for ultra sonic sensor
62                        rLine :: Line, -- Line for line follower sensor
63                        rLEDs :: (LED, LED) -- 0 for LED 1, 1 for LED 2
64                      } deriving (Eq, Ord, Show)
65
66 -- A world contains a robot, walls and lines
67 -- As extra it stores the viewport size of the source file and mouse position
68 data World = World { wRobot     :: Robot
69                    , wWalls    :: [G.Point]
70                    , wLines    :: [Seg]
71                    , wBorder :: G.Point -- Max (x, y) in text representation
72                    , wMousePos :: G.Point
73                    } deriving (Eq, Ord, Show)
74
75 -- The command constructors are pattern matched by executeCommand
76 data Command = SetMotor Float Float |      -- vl vr
77                SetRGB Int Int Int Int |    -- index r g b
78                PlayTone Int Int
79
80 -- The simulator needs 3 channels for communication via the exported interface
81 -- One MVar for recieving commands, two for transmitting the sensor data
82 data Device = Device { dCommand :: MVar Command,
83                        dDistance :: MVar Float,
84                        dLine :: MVar Line }
85
86 data Line = LEFTB | RIGHTB | BOTHB | BOTHW deriving (Eq, Ord, Show)
87
88 lineFromBools (True , False) = LEFTB
89 lineFromBools (False, True ) = RIGHTB
90 lineFromBools (True , True ) = BOTHB
91 lineFromBools (False, False) = BOTHW
92
93 -- EXPORTED MBOT FUNCTIONS --
94 openMBot :: IO Device
95 openMBot = do world     <- readWorld
96 "/Users/berombau/FunProg/src/simulator_grid.txt"
97               [rp,wp]   <- mapM loadBMP [
98 "/Users/berombau/FunProg/src/robotOFF.bmp"
99                                         , "/Users/berombau/FunProg/src/wall.bmp"]
100              c         <- newEmptyMVar
101              d         <- newEmptyMVar
```

```
102               l         <- newEmptyMVar
103               -- print "Forking Gloss"
104               forkIO $ G.playIO initWindow -- display
105                  G.white                   -- background
106                  30                        -- fps
107                  world                     -- initial world
108                  (render rp wp)            -- render world
109                  handleEvent               -- handle input
110                  (simulateWorld (Device c d l))
111               -- print "Returning Device from Simulator"
112               return (Device c d l)
113
114 closeMBot :: Device -> IO ()
115 closeMBot d = return ()
116
117 sendCommand :: Device -> Command -> IO ()
118 sendCommand Device { dCommand = mvar } = putMVar mvar
119
120 readUltraSonic :: Device -> IO Float
121 readUltraSonic Device { dDistance = mvar } = takeMVar mvar
122
123 readLineFollower :: Device -> IO Line
124 readLineFollower Device { dLine = mvar } = takeMVar mvar
125
126 setMotor :: Float -> Float -> Command
127 setMotor = SetMotor
128
129 setRGB :: Int -> Int -> Int -> Int -> Command
130 setRGB = SetRGB
131
132 playTone :: Int -> Int -> Command
133 playTone = PlayTone
134
135
136 -- GEOMETRY FUNCTIONS --
137
138 -- Returns (dX, dY) for dt and Robot
139 nextPosition :: Float -> Robot -> G.Point
140 nextPosition dt Robot { rCoord = (x, y),
141                         rVl = vl,
142                         rVr = vr,
143                         rWr = wr,
144                         rTheta = theta } =
145     (x + (dt * (wr / 2) * (vl + vr) * cos theta),
146      y + (dt * (wr / 2) * (vl + vr) * sin theta))
147
148 -- Returns dTheta for dt and Robot
149 nextDirection :: Float -> Robot -> Float
150 nextDirection dt Robot { rCoord = (x, y),
151                          rVl = vl,
152                          rVr = vr,
153                          rWr = wr,
154                          rWa = wa,
155                          rTheta = theta } =
```

```
156     normalizeAngle $ theta + (dt * (wr / wa) * (vr - vl))
157
158 -- Returns (left, right) points of sensors, offset from Robot point by
159 distLineSensors
160 getLineSensorPoints :: Float -> G.Point -> Float -> Seg
161 getLineSensorPoints size p theta =
162     mapTuple (+ p) (mulSV (distLineSensors * size / 2) (cos (pi - theta), sin
163 theta),
164                     mulSV (distLineSensors * size / 2) (cos theta       , sin (-
165 theta)))
166
167 -- Returns the segment representing the line of sight of the distance sensor
168 getSightLine :: Float -> G.Point -> Float -> Seg
169 getSightLine size p theta = (p + mulSV (maxDistSensor * size) (cos theta, sin
170 theta), p)
171
172 getSegments :: Float -> G.Point -> [Seg]
173 getSegments s (x, y) = [((x + s, y + s), (x + s, y - s)),
174                         ((x - s, y - s), (x + s, y - s)),
175                         ((x - s, y - s), (x - s, y + s)),
176                         ((x + s, y + s), (x - s, y + s))
177                        ]
178 -- s is (cell / 2) when calculated on pixel points, 0.5 when on grid points
179
180 -- Returns [ 4 points ] of the rectangle for a segment and const lineThickness
181 getRectFromSeg :: Seg -> Path
182 getRectFromSeg (p1, p2) = [ transform lineThickness p1,
183                             transform (-lineThickness) p1,
184                             transform lineThickness p2,
185                             transform (-lineThickness) p2 ]
186   where transform dy p = rotateV (argV p) $ offset dy $ rotateV (-(argV p)) p
187         offset y' (x, y) = (x, y + y')
188
189 -- Returns distance between two points
190 distance :: G.Point -> G.Point -> Float
191 distance (x1 , y1) (x2 , y2) = sqrt (x'*x' + y'*y')
192     where
193       x' = x1 - x2
194       y' = y1 - y2
195
196
197 -- UTILITY FUNCTIONS --
198 mapTuple = join (***)
199
200
201 -- GLOSS FUNCTIONS --
202
203 -- Given a list of lines, create a world. In this string:
204 -- - > is the position of the robot.
205 -- - X is the position of a wall.
206 --
207 -- For example, in:
208 --
209 --     x
```

```
--    -------------------->
-- y  +--------------------+
--    | |                  |
--    | |  X    >          |
--    | |  X               |
--    | |  XXXXXXXXXX      |
--    | |           X      |
--    | |           X      |
--    v +--------------------+
-- (0,0) (10,10)
-- (10,10) (20,10)

-- we have a robot on (2, 2), walls on (1, 2) and (1, 3) and so on.
-- two line segment at ((0,0), (10,10)) and ((10,10), (20,10))
makeWorld :: [String] -> World
makeWorld ls = addBorder $ addLines $ foldr (uncurry addPiece) emptyWorld
(withCoords grid)
  where withCoords grid = [(chr, (c, r))
                           | (r, row) <- zip [0..] (reverse grid)
                           , (c, chr) <- zip [0..] row
                           ]
        addPiece '^' coord world = setRobot (pi/2)     coord world
        addPiece '>' coord world = setRobot 0          coord world
        addPiece 'v' coord world = setRobot (3/2 * pi) coord world
        addPiece '<' coord world = setRobot pi         coord world
        addPiece 'X' coord world = world { wWalls = coord:wWalls world  }
        addPiece _ _ world = world

        setRobot theta coord world = world {
                 wRobot = (wRobot world) { rTheta = theta, rCoord = coord }
                 }

        addBorder world = world {
                 wBorder = snd $ maximumBy (comparing (\(_, (x, y)) -> x + y))
(withCoords grid)
                 }

        (grid, lineList) = span (\l -> head l `elem` "+|") ls
        addLines world = world { wLines = map (getLine.take 2.words) lineList }
        getLine :: [String] -> Seg
        getLine [ _:p1, _:p2 ] = ((read (getFirst $ init p1), read (getSecond $
init p1))
                                 , (read (getFirst $ init p2), read (getSecond $
init p2)))
        getFirst :: String -> String
        getFirst = takeWhile (/= ',')
        getSecond :: String -> String
        getSecond = tail . dropWhile (/= ',')

-- Given a filename, read and return a list of worlds.
readWorld :: String -> IO World
readWorld f = makeWorld . lines <$> readFile f

-- Move the player in given direction if the player can move in this direction.
```

```haskell
-- A player can move in a direction iff the cell in this direction:
-- - is empty; or
-- - is a storage cell; or
-- - contains a crate and the cell behind it is empty; or
-- - contains a crate and the cell behind it is a storage cell.

render :: G.Picture -> G.Picture -> World -> IO G.Picture
render rp wp gridworld = return $ G.pictures $
                map (G.polygon.getRectFromSeg) ls
             ++ map renderLines ls
             ++ map (renderPicAt wp) w
             ++ map renderLines (concatMap (getSegments (cell / 2)) w)
             ++ [renderPicAt (transformRobot $ G.pictures [drawLEDs rp r]) (rCoord r)]
             ++ [G.color (makeColorI 0 255 0 255 ) $ G.Line [sx, sy]]
             ++ map renderLines [getSightLine cell (rCoord r) (rTheta r)]
             ++ [ renderDebugText (-500, -200) $ G.Text $ show $ wBorder world,
                  renderDebugText (-500, -220) $ G.Text $ show $ wMousePos world,
                  renderDebugText (-500, -240) $ G.Text $ show $ wRobot gridworld,
                  renderDebugText (-500, -260) $ G.Text $ show $ wLines gridworld
                ]

    where r = wRobot world
          w = wWalls world
          ls = wLines world
          world = toPixelCoord gridworld
          toPixelCoord:: World -> World
          toPixelCoord w = w { wRobot = r, wWalls = ws, wLines = ls }
            where r = (wRobot w) { rCoord = correctPoint (rCoord (wRobot w)) }
                  ws = map correctPoint (wWalls w)
                  ls = map correctLine (wLines w)
                  correctLine = mapTuple correctPoint
                  correctPoint :: G.Point -> G.Point
                  allCoord = wWalls w ++ [rCoord (wRobot w)] ++ concatMap (\(p1, p2) -> [p1, p2]) (wLines w)
                  size which = maximum $ map which allCoord
                  toPix which = (+ (cell / 2 - cell * which (wBorder w) / 2))
                                . (* cell)
                  correctPoint (x, y) = (toPix fst x - maxX / 2, toPix snd y - maxY / 2)
                  (maxX, maxY) = wBorder w
          renderDebugText (x, y) p  = renderPicAt (G.scale 0.1 0.1 p) (x, y)
          (sx, sy) = getLineSensorPoints cell (rCoord r) (-(rTheta r))
          renderPicAt picture (x, y) = G.translate x y picture
          -- G.rotate does clockwise rotation, so a negative angle will
          transformRobot = G.rotate $ radToDeg $ (- rTheta r) + pi / 2
          drawLEDs rp Robot { rLEDs = (l1, l2) } = G.pictures [rp, drawLED l1 (-15, 15), drawLED l2 (15, 15)]
          drawLED (r, g, b) (x, y) = G.Color (makeColorI r g b 255) $ G.translate x y $ G.rectangleSolid 10 5
          renderLines :: Seg -> G.Picture
          renderLines (p1, p2) = G.Line [p1, p2]

handleEvent :: G.Event -> World -> IO World
```

```haskell
handleEvent (G.EventKey (G.MouseButton G.LeftButton) G.Down _ (xPos, yPos)) =
handle
  where handle :: World -> IO World
        handle wld = return wld { wMousePos = (xPos, yPos) }
handleEvent (G.EventKey (G.SpecialKey key) G.Up _ _) = handle key
  where handle :: G.SpecialKey -> World -> IO World
        handle G.KeyDown   wld = setWheelSpeed (-defaultSpeed) (-defaultSpeed) wld
        handle G.KeyUp     wld = setWheelSpeed defaultSpeed defaultSpeed wld
        handle G.KeyLeft   wld = setWheelSpeed (-1) 1 wld
        handle G.KeyRight  wld = setWheelSpeed 1 (-1) wld
        handle G.KeyDelete wld = setWheelSpeed    0 0 wld
        handle G.KeySpace  wld = changeLocation (0, 0) wld
        handle _           wld = return wld
        changeLocation coord world@World { wRobot = robot } = return world {
wRobot = robot { rCoord = coord } }
        changeWheelSpeed :: Float -> Float -> World -> IO World
        changeWheelSpeed l r world@World { wRobot = robot@Robot { rVl = wl, rVr =
wr }} = return world { wRobot = robot { rVl = wl + l, rVr = wr + r }}
        setWheelSpeed l r world = return world { wRobot = (wRobot world) { rVl =
l, rVr = r }}
handleEvent _ = return

simulateWorld :: Device -> Float -> World -> IO World
simulateWorld dev timeStep world = applyDevToWorld dev $ adjustDistance $
adjustRLine world { wRobot = moveRobot $ wRobot world }
        where r = wRobot world
              moveRobot :: Robot -> Robot
              moveRobot r = r { rCoord = nextPosition timeStep r,
                                rTheta = nextDirection timeStep r
                              }
              adjustDistance :: World -> World
              adjustDistance w = w { wRobot = rob { rDistance = calculateDistance
(wWalls w) (getSightLine 1 origin theta) origin } }
                 where rob = wRobot w
                       theta = rTheta rob
                       origin = rCoord rob
              calculateDistance :: [G.Point] -> Seg -> G.Point -> Float
              calculateDistance ws (p0, p1) origin = delimiter $ mapMaybe mapper
(toSegments ws)
                 where mapper :: Seg -> Maybe G.Point
                       mapper (p2, p3) = intersectSegSeg p0 p1 p2 p3
                       toSegments :: [G.Point] -> [Seg]
                       toSegments = concatMap (getSegments 0.5)
                       delimiter :: [ G.Point ] -> Float
                       delimiter [] = maxDistSensor
                       delimiter ls = minimum (map (distance origin) ls)
              adjustRLine :: World -> World
              adjustRLine w = w { wRobot = (wRobot w) { rLine = calculateRLine
(wLines w) } }

              calculateRLine :: [Seg] -> Line
              calculateRLine ls = lineFromBools $ mapTuple mapper sensorSeg
                 where mapper :: G.Point -> Bool
                       -- Test if point
```

```
372                         sensorSeg@(sp1, sp2) = getLineSensorPoints 1 (rCoord r)
373 (rTheta r)
374                         mapper p = any testIfInBox (map testIfOnLine ls)
375                         testIfInBox (Just _, (lp1, lp2)) = distance p
376 (closestPointOnLine lp1 lp2 p) < lineThickness / 2
377                         testIfInBox (Nothing, _) = False
378                         p = rCoord r
379                         r = wRobot world
380                         testIfOnLine l@(p1, p2) = (intersectSegSeg p1 p2 sp1 sp2,
381 l)
382
                testInBox :: G.Point -> (G.Point, G.Point) -> Bool
                testInBox p0 (p1, p2) = pointInBox p0 p1 p2

                size which = maximum $ map which (wWalls world)
                toPix which = (+ (cell / 2 - cell * size which / 2))
                            . (* cell)
                renderPicAt picture (x, y) = G.translate (toPix fst x)
                                                         (toPix snd y)
                                                         picture

                applyDevToWorld :: Device -> World -> IO World
                applyDevToWorld (Device command distance line) w = do
                       maybeCommand <- tryTakeMVar command
                       _ <- tryTakeMVar distance
                       _ <- tryPutMVar distance (rDistance (wRobot world))
                       _ <- tryTakeMVar line
                       _ <- tryPutMVar line (rLine (wRobot world))
                       executeCommand maybeCommand w

executeCommand :: Maybe Command -> World -> IO World
executeCommand Nothing w = return w
executeCommand (Just (SetRGB i r g b)) w@World { wRobot = rob } = return w {
wRobot = rob { rLEDs = changeLED (rLEDs rob) i r g b } }
     where  changeLED :: (LED, LED) -> Int -> Int -> Int -> Int -> (LED, LED)
            changeLED (_, l) 1 r g b = ((r, g, b), l)
            changeLED (l, _) 2 r g b = (l, (r, g, b))
executeCommand (Just (SetMotor vl vr)) w@World { wRobot = rob } = return w {
wRobot = rob { rVl = vl, rVr = vr } }
executeCommand (Just _) w = return w
```