Functional design of a simple board game

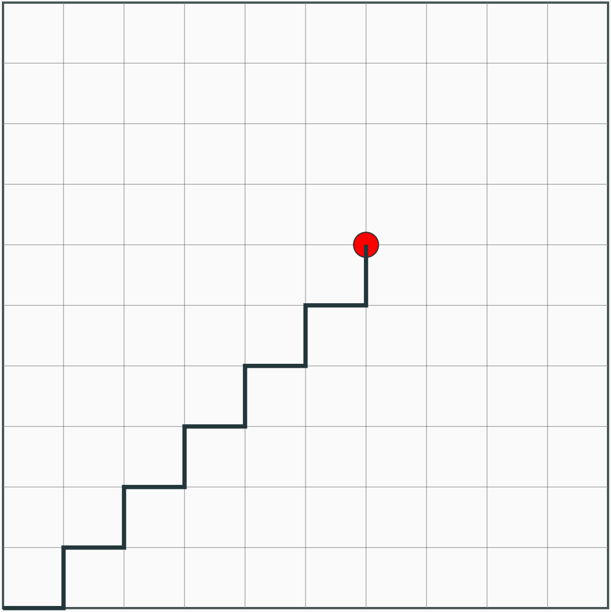
Alessandro Candolini

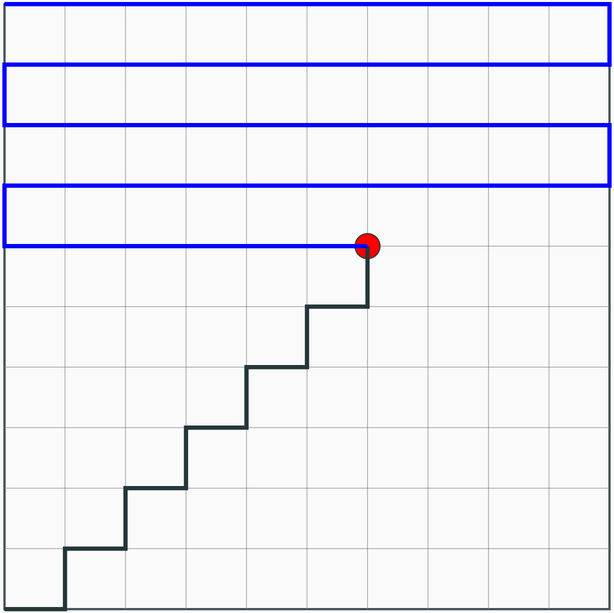
March 14, 2023

Agenda

1. Warm up
2. Algebra of moves
3. Board
4. Algebra of strategies
5. Advanced considerations

Warm up





One strategy is described in words as follows:

	move right until the edge
and then	move one step down
and then	move left until the edge
and then	move one step down
and then	repeat

repeat

	fill right
and then	one step down
and then	fill left
and then	one step down

```
nextMoveH :: Position -> Maybe Move
nextMoveH (Pos x y) =
  if y `mod` 2 == 0 then
     (if x < 19 then
         Just RightM
      else (
         if y < 19 then Just DownM
         else Nothing)
     )
  else
     (if x > 0 then
         Just LeftM
      else (
         if y < 19 then Just DownM
      else Nothing)
     )
```

```
nextMoveH :: Int -> Position -> Maybe Move
nextMoveH n (Pos x y) =
  if y `mod` 2 == 0 then
     (if x < (n-1) then
         Just RightM
      else (
         if y < (n-1) then Just DownM
         else Nothing)
     )
  else
     (if x > 0 then
         Just LeftM
      else (
         if y < (n-1) then Just DownM
      else Nothing)
     )
```

Add epic / PER-1

Implement horizontal strategy

Attach | Add a child issue | Link issue

Description

fill row right

and then one step down

and then fill row left

and then one step down

Activity

Show: All | Comments | History   Newest first

Add a comment...

Pro tip: press M to comment

App.hs M

src > App.hs

```haskell
nextMoveH :: Position -> Maybe Move
nextMoveH (Pos x y) =
  if y `mod` 2 == 0 then
      (if x < 19 then
          Just RightM
       else (if y < 19 then
                Just DownM
             else Nothing)
      )
  else (
    if x > 0 then
        Just LeftM
    else (if y < 19 then
            Just DownM
          else Nothing)
  )

```

fill row right
and then one step down
and then fill row left
and then one step down

```
nextMoveH :: Int -> Position
   -> Maybe Move
nextMoveH n (Pos x y) =
  if y `mod` 2 == 0 then
     (if x < (n-1) then
         Just RightM
      else (
         if y < (n-1) then
           Just DownM
         else Nothing)
     )
  else
     (if x > 0 then
         Just LeftM
      else (
         if y < (n-1) then
           Just DownM
         else Nothing)
     )
```

- hypersensitivity to details (eg, choice of coordinates)
- focus on *operational* concerns (i. e., the *how*) vs *denotational* concerns (i. e., the *what*)
- impedence mismatch between acceptance criteria and implementation, obfuscating the aim / behaviour of the code (ie, need to reverse engineering the code to understand the requirements)
- lack of composability

fill right

and then one step down

and then fill left

and then one step down

fill right
andThen oneStep down
andThen fill left
andThen oneStep down

```
horizontal = repeat $ fill right
   `andThen` oneStep down
   `andThen` fill left
   `andThen` oneStep down
```

```
vertical = repeat $ fill down
    `andThen` oneStep right
    `andThen` fill up
    `andThen` oneStep right
```

Algebra of moves

```
data Position = P Int Int deriving (Eq,Show)
```

```haskell
import Prelude hiding (Right, Left)

data Direction = Left | Right | Up | Down
         deriving (Eq, Show, Enum, Bounded)
```

Algebra of moves

```
data Move

step :: Direction -> Move

runMove :: Move -> Position -> Position
```

```
left = step Left
right = step Right
up = step Up
down = step Down
```

```
runMove right (P 1 1) `shouldBe` (P 2 1)
runMove left (P 1 1) `shouldBe` (P 0 1)
runMove up (P 1 1) `shouldBe` (P 1 2)
runMove down (P 1 1) `shouldBe` (P 1 0)
```

Internal representation: option A

```
data Move = Move {
   runMove :: Position -> Position }
```

```
step :: Direction -> Move
step Right = Move $ \p -> case p of
   P x y -> P (x+1) y
...
```

Internal representation: option B

```
data Move = Step Direction
    derives (Eq, Show)
```

```
step :: Direction -> Move
step = Step
```

```
runMove :: Move -> Position -> Position
runMove (Step Right) (P x y) = P (x + 1, y)
...
```

Extend the algebra

```
upRight = up <> right
twoUpOneRight = up <> up <> right
```

```haskell
instance Semigroup Move where
  m1 <> m2 = ...

instance Monoid Move where
  mempty = ...
```

```haskell
newtype Move = Move  {
   runMove :: Position -> Position }

instance Semigroup Move where
  (Move m1) <> (Move m2) = Move ( m1 . m2 )

instance Monoid Move where
  mempty = Move id
```

```haskell
{-# LANGUAGE DerivingVia #-}

newtype Move = Move  {
   endo :: Endo Position } deriving
     (Semigroup, Monoid) via (Endo Position)

runMove :: Move -> Position -> Position
runMove = appEndo . endo
```

Internal representation: option B

```
data Move = DontMove
    | Step Direction
    | Compose Move Move derives (Eq, Show)

instance Semigroup Move where
  m1 <> m2 = Compose m1 m2

instance Monoid Move where
  mempty = DontMove
```

```haskell
simplify :: Move -> Move
simplify DontMove = DontMove
simplify s@(Step d) = s
simplify (Compose m1 m2) =
  case (simplify m1, simplify m2) of
    (DontMove, m) -> m
    (m, DontMove) -> m
    (Step Left, Step Right) -> DontMove
    (Step Right, Step Left) -> DontMove
    (p1, p2) -> Compose p1 p2
```

Notice: This implementation does not take into account associativity

```
instance Group Move where
  invert Stay = Stay
  invert (Step Right) = Step Left
  invert (Step Left) = Step Right
  invert (Step Up) = Step Down
  invert (Step Down) = Step Up
  invert (Combine m1 m2) =
    Combine (invert m2) (invert m1)
```

Board

The board

```
data Board

square :: Int -> Board
infinite :: Board
walls :: [Position] -> Board
oneStep right `andThen` oneStep down
check :: Board -> Position -> Availability

data Availability = Available | Unavailable
   deriving (Eq,Show)
```

```
data Board = Board {
   check :: Position -> Availability }
```

```
square n = Board $ \p -> ...
```

```haskell
data Board = Infinite | SquareBoard Int
   deriving (Eq,Show)

square = SquareBoard

check (SquareBoard n) (P x y) = ...

draw :: Board -> Text
```

Algebra of strategies

Algebra of strategies

```
data Strategy

oneStep :: Move -> Strategty

run :: Strategy -> Board -> Position -> [Position]
```

```
run (oneStep right)
    (squareBoard 10)
    (P 0 0) `shouldBe` [(P 1 0)]
```

```
run (oneStep left)
    (squareBoard 10)
    (P 0 0) `shouldBe` []
```

We might model the failures more explicitly later, in the interest of

- Players who fail at a earlier step
- Model more precisely strategies that can recover from failures (eg, fill) vs strategies that cannot (eg, single move)

```
data Strategy = OneStep Move deriving (Eq,Show)

oneStep = OneStep

run (OneStep m) b p = ...
```

An helper method

```haskell
runMove :: Move -> Position -> Position -- runner

-- board-aware modified runner
move :: Board -> Move -> Position -> Maybe Position
move b m =
      mfilter isAvailable
      . pure
      . (runMove m)


isAvailable = (Available ==)  . check b
```

Suggestive considerations for another time:

- the board acts by "deforming" / "curving" the interpreter `runMove` (interpreter in a modified "spacetime")
- the original behaviour of `runMove` can be recovered by using the infinite board with no walls
- semantically, we can argue that a runner of move makes no sense without a board

This possibly suggests we should look at ways to make the relationship between moves and boards closer (eg, a board *is* a runner of a move)

For now let's move forward with our ad-hoc `move` method

```
run (OneStep m) b = maybeToList . move b m
```

```
s = oneStep right
    `andThen` oneStep down
    `andThen` oneStep left
    `andThen` oneStep up

run s (squareBoard 10) (P 0 0)
   `shouldBe` [(P 1 0), (P 1 1), (P 0 1), (P 0 0)]
```

Snake style:

```
s = oneStep right
    `andThen` oneStep down
    `andThen` oneStep left
    `andThen` oneStep up

run s (squareBoard 10) (P 0 0)
   `shouldBe` [(P 0 0), (P 0 1), (P 1 1), (P 1 0)]
```

```
andThen :: Strategy -> Strategy -> Strategy
```

```
data Strategy = OneStep Move
  | AndThen Strategy Strategy deriving (Eq,Show)

oneStep = OneStep
andThen = AndThen

run (OneStep m) b = maybeToList . move b m
run (AndThen s1 s2) b p = run2 . run1 where
   run1 = run s1 b
   run2 [] = []
   run2 t@(h:_) = run s2 b h ++ t
```

```
s = oneStep up
    `andThen` oneStep right

run s (squareBoard 10) (P 9 0)
   `shouldBe` [(P 9 1)]
```

```
s = oneStep right
    `andThen` oneStep up

run s (squareBoard 10) (P 9 0)
   `shouldBe` []
```

```
s = oneStep right
    `andThen` oneStep down
    `andThen` oneStep left

run s (squareBoard 10) (P 8 0)
   `shouldBe` [(P 9 0)] ???
```

```
s = (oneStep right
    `andThen` oneStep down)
    `andThen` oneStep left

run s (squareBoard 10) (P 8 0)
   `shouldBe` [(P 9 0)] ???
```

```
run' :: Strategy -> Board -> Position -> (Maybe Positi
```

Advanced considerations

- random generation
- walls
- self avoiding (single player)
- self avoiding (multiplayer)
- UI + latex output

Questions?