Functional design of a simple board game

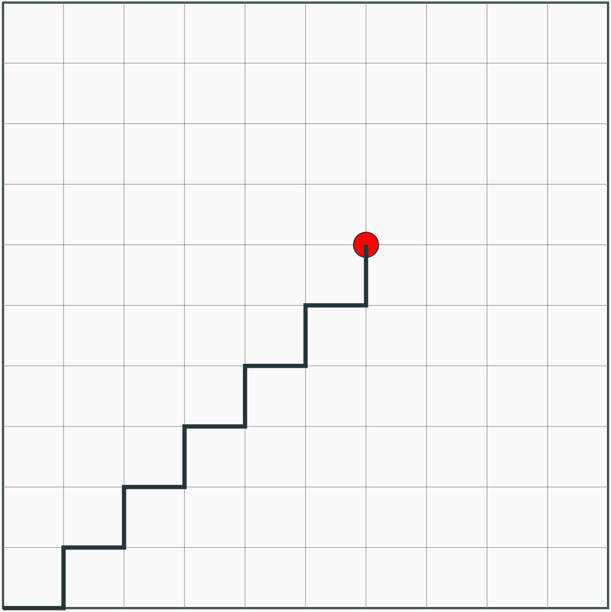
Alessandro Candolini

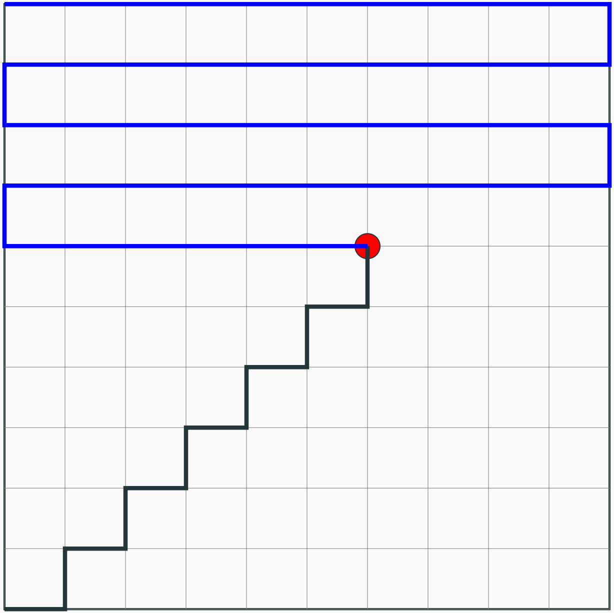
March 12, 2023

Agenda

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

Warm up





One strategy is described in words as follows:

move right until the edge

then move one step down

then move left until the edge

then move one step down

then repeat

fill row right

and then one step down

and then fill row left

and then one step down

repeat

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

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

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

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

```
data Move

step :: Direction -> Move

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

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

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

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

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

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

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

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

instance Monoid Move where
  mempty = DontMove
```

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

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

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

```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

```haskell
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)
```

Algebra of strategies

```haskell
data Board

rectangular :: Int -> Int -> Board

square n = rectangular n n

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

check :: Board -> Position -> Availability
```

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

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

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

draw :: Board -> Text
```

Advanced considerations

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

Questions?