Iterated Prisoner Dilemma - a Haskell implementation

December 13, 2019

Abstract

In this report we outline how to define and implement the iterated prisoner dilemma in the programming language Haskell. Haskell was chosen because working with lists and pure functions are two of the languages' strengths. The expressiveness of the type system makes it easier to implement, test and prove the code correct. The iterated prisoner dilemma is specified as a set of tournaments which are in turn a set of iterations of games players. Each player can only decide to cooperate or defect in a given game based on the players games' history and does not have access to the information of its opponent current decision.

1 Miscellaneous and Setup

We begin by enabling a compiler extension to write `Show` and `Eq` instances for functions. This keeps parts of the code simpler.

```
{-# LANGUAGE FlexibleInstances #-}
```

Because the file can be run as a standalone program we have to define the `module`

```
module Main where
```

Next we import a couple of useful functions from Haskell standard libraries.

```
import System.Random
import Data.List (nubBy, sortBy, intercalate)
import Data.Bifunctor (bimap)
import Data.Function (on)
```

We need `System.random` to generate lists of random numbers to shuffle the player populations. The `Data.List` and `Data.Bifunctor` imports are useful functions to keep the code clean and concise. `bimap` is used to efficiently work on data in tuples, as we store loads of information in them. An example how `bimap` works can be seen here:

```
bimap (*5) (show) (1,1) == (5,"1")
```

An example of how the project will be used is given by the `main` function. (The details for the functions used in `main` are given in the subsequent document.)

```
main :: IO ()
main =  do
  result <- startSimulation 100 3 100
  print $ show $ stats $ snd result
```

Yielding an output similar to the following:

```
Simulating Iterated prisoner
Population [("Defector",25),("Cooperator",25),("TFT",25),("Yasha",25)]
[("Defector",18),("Cooperator",23),("TFT",31),("Yasha",27)]
```

The first line is just to show the user that the code has started running, because depending on the input sizes, it could take some time to complete. The second line is statistics for the population, meaning there are `x` many of Player-Type `"Name"` and the third line is the statistics for the final population after the program completed. You can then go on and compare the populations to each other.

2 Types and Declarations

In most comprehensive and well written functional programs one starts out with defining the data structures. This is done because correct/meaningful data structures guide and aide the implementation of needed functions. The most basic action/state we need to model in the program is a players `Choice` for some confrontation. In the iterated prisoner a player can choose to cooperate or defect in a confrontation.

```
data Choice = Cooperate | Defect
     deriving (Eq, Show)
type BattleResult = (Choice, Choice)
```

We also specify that a `BattleResult` is a tuple of two choices (of two players respectively). With that defined we can define how a player looks like:

```
type PlayerID = Int
type Payment = Int
type PlayerHist = [((Choice, Payment), (PlayerID, Choice))]
data Player = Player
                  { name :: String -- strategy name
                  , playerID :: PlayerID
                  , decide :: PlayerHist -> Choice
                  , getPlayerHist :: PlayerHist
                  }
```

The type definitions are just aliases. Playerhist is our first more complex type:

The history for a player is defined as a list (of interactions) where each interaction consists of a the own choice and the related payment ((Choice, Payment)) and the opponents' PlayerID and Choice ((PlayerID, Choice)). we need the first part to determine how much payment a player received during a set of games and the second part for the player to determine the choice by the given strategy. The PlayerID is stored solely for manual inspection of game outcomes.

A Player is then defined as having a name (the name of the strategy the player pursues) an ID for inspection, a strategy (decide, which is a function that generates a Choice when given a player history) and her/his own history (getPlayerHist[1]).

We also defined Show and Eq instances for Player as well as for the type Int→Player[2].

```
instance Show Player where
  show (Player n p _ o) =
    "Player { name: '" ++ n ++ "'" ++
           ", playerID: " ++ (show p) ++
           ", getPlayerHist: " ++ (show o) ++ "'}"

instance Eq Player where
  (Player n _ _ _) == (Player n' _ _ _) = n == n'

instance Eq (Int -> Player) where
  p1 == p2 = (p1 0) == (p2 0)

instance Show (Int -> Player) where
  show p = show $ p 0
```

As we need not only a single player, but a whole population of them we define

```
type Population = [Player]
```

as a synonym.

There are two more type we are using in the code:

```
type RandList        = [Int]
type IterationResult = [Player]
```

[1]The naming getPlayerHist was chosen to make some code more comprehensible and has no special meaning.

[2]Int→Player is the type of the player generator function which constructs a player with a given playerID. As you will see there are a couple of instances of this data type in the code and by using these instances we get around the hassle of having to "produce" players before showing or comparing them. (Showing and comparing players don't use the playerID argument, so we can work without them.)

The first one being the alias of a list containing random integers[3]. Latter being the result of one iteration[4]. It is no coincidence that `IterationResult` is defined in the same way as `Population`; that way, the result of one iteration (a new population with a different distribution of players) can be used to start a new one. The last declaration we are making is the coded variant of the payout matrix for this example:

```haskell
payment :: BattleResult -> (Int, Int)
payment (Cooperate, Cooperate) = (3,3)
payment (Cooperate, Defect)    = (1,4)
payment (Defect, Cooperate)    = (4,1)
payment (Defect, Defect)       = (2,2)
```

Given a `BattleResult` this function calculates the payouts for both players respectively[5].

3 Player Configuration

The code in this section describes the available player types for this example. Being hardcoded it would nevertheless be easy to abstract it to a more general form where the user can configure the players. As a matter of fact, if the user can instantiate `Player`s the requirements for this abstraction are already met. The following `Player`s were hardcoded more for convenience and less for the need of it:

[3]We assume this list to be infinite; however, the code is written to handle calls with finite lists of random numbers.

[4]The definitions of these terms are given in the abstract and the introduction.

[5]The reason for hardcoding the matrix was that the code was written for a specific example and not for general purpose use. It would be easy however to remove these dependencies and abstract it in a way that can handle more user configuration.

```haskell
defector :: Int -> Player
defector n = Player
                "Defector"
                n
                (\_ -> Defect)
                []

cooperator :: Int -> Player
cooperator n = Player
                "Cooperator"
                n
                (\_ -> Cooperate)
                []


tftDecide :: PlayerHist -> Choice
tftDecide []             = Cooperate
tftDecide ((_,(_,c)):_) = c

tft :: Int -> Player
tft n = Player
                "TFT"
                n
                tftDecide
                []

rageDecide :: PlayerHist -> Choice
rageDecide [] = Cooperate
rageDecide l  = if (elem Defect . map getOpChoice $ l) then Defect else Cooperate
                  where getOpChoice = snd . snd

rage :: Int -> Player
rage n = Player
                "Yasha"
                n
                rageDecide
                []
```

Let’s take a look at the last player as it is the most complicated one: The player can be constructed using the function `rage`. The given Integer is used as the players ID. The players name is ”Yasha” and the players history is empty on instantiation, becase the player was not part of any confrontations. The function `rageDecide` is the players strategy or decision function. As the type `Player` enforces the functions’ type is `PlayerHist` →`Choice`. Players with rage strategies are considered to `Cooperate` in the first battle (so if the history is empty) and only `Cooperate` in subsequent confrontations if their opponent has never `Defect`ed.

The function

```
elem Defect ∘ map getOpChoice $ l
```

retrieves the opponents choices from the history (as `getOpchoice` suggests) and evaluate to true if `Defect` is an `element` in the resutling list.

For convenience we also defined

```
playerTypes :: [Int -> Player]
playerTypes = [defector, cooperator, tft, rage]
```

as the different types of players that are available during the game. Defining it as a constant in the code keeps us from constructing the array again and again throughout the following code.

Finally, we defined a function that generates a Population, given a distribution:

```
generatePopulation :: [(Int->Player, Int)] -> Population
generatePopulation = map (\(i,p) -> p i) .
                           zip [1..] .
                           intercalate [] .
                           map (\(p,n) -> replicate n p)
```

The distribution is given by a list of player generators tupled with the corresponding player count in the resulting population. This is done by first constructing a list of lists of generator functions. Semantically this corresponds to a list of n player constructors for n corresponding players in the population. `intercalate []` flattens this list (combines the sublists to one). The output equivalent to a population constructor, that is zipped with integers that are subsequently used to instantiate the players.

4 Game Logic

Before we jump into the code, a quick refresher on how the game logic works: A tournament is a set of iterations of individual games. Meaning that in one tournament, x games are played by the same (initially randomly drawn) individulas. After one such tournament, we calculate the overall payout and construct a new population with a distribution matching a player types' combined payout[6]. This new population has the same size as the old one and is then used for the next tournament.

First we take a look at the `play` function which defines the interations:

[6]The payout of individuals of the same type e.g. rage players are summed up.

```
--                 shuffled population  iteration count
runIteration :: Population ->        Int ->           IterationResult
runIteration p i = undoPairs $ play i (makePairs p)

--       counter   shuffled list of battles
play :: Int ->    [(Player, Player)] ->    [(Player, Player)]
play 0 h         = h
play i p
    | i < 0      = p
    | otherwise = play (i-1) $ newPlayers decisions
  where
    dec p = decide p $ getPlayerHist p
    decisions = zip p $ map (bimap dec dec) p :: [((Player, Player), BattleResult)]
    newPlayers =
      map (\((p1,p2),cs@(c1,c2)) ->
             let (a1, a2) = payment cs
             in
             (p1{getPlayerHist = ((c1, a1),(playerID p2, c2)):(getPlayerHist p1)}
             ,p2{getPlayerHist = ((c2, a2),(playerID p1, c1)):(getPlayerHist p2)}))
```

`runIteration` is only a wrapper function that hides the pair drawing and back conversion of those pairs from the other implementations. Walking through the `play` function itself is rather straight forward: if the number of iterations $i \leq 0$ the current population state is returned. Otherwise play is called again with $i - 1$ and an updated population[7]. The update of the population is done in two steps:

1. The individual players decisions are calculated using the players own decide functions[8]. To remind you, the decide functions themselves are of type `[((Choice, Payment), (PlayerID, Choice))]`. With that information each player can decide on their own. The result is zipped together with the player pair to get a list of `((Player, Player), (Choice, Choice))` where the choices belong the the respective player.

2. The players histories are updated using their made decisions. This happens in the `newPlayers` function which, for every pair of the before mentioned type adds a new entry to each players history[9]. The : operator appends the new history entry to the existing histories.

Having the logic for the iterations defined, we can define the tournament logic whose purpose is to run the iteration n times, construct new populations with different distributions from the results and shuffle the populations each

[7]The history of the individual players is updated to include the new `Choice`es.

[8]bimap applies the function `dec` to both entries in a tuple

[9]The notation `p1{getPlayerHist =○..}` is called record notation and is just the construction of a new player with the same properties as the old one except for the assignment made in the record field `{...}`.

time to get meaningful results. The code for `runGame` encapsulates this logic and is probably the most complex one in the program:

```
--          tournaments  maxIterations  initial Population
--            for shuffling   stats for tournaments         with updated histories
runGame :: Int ->       Int ->         ([[(Int->Player, Int)]], Population) ->
              RandList ->    ([[(Int->Player, Int)]],       Population)
runGame _ maxIter res [] = res
runGame 0 maxIter res _  = res
runGame i maxIter res@(hist,ps) rs@(h:t)
  | i < 0           = res
  | otherwise       = runGame (i-1) maxIter (iterStats:hist, newPopulation) $
                        drop (length iteration) t
  where
    getPayments = map (snd . fst) . getPlayerHist      :: Player -> [Payment]
    iteration   = runIteration (shuffle rs ps) maxIter :: Population
    iterStats   = map (\p -> (p, sum .
                              map (sum . getPayments) .
                              filter (==(p 0)) $ iteration)
                        ) playerTypes :: [(Int->Player, Payment)]
    payments    = sum . map snd $ iterStats
    newPopulationStats = map
                          (\(p, s) -> (p, calcCount s payments (length ps)))
                          iterStats  :: [(Int->Player, Payment)]
    newPopulation      = generatePopulation newPopulationStats  :: [Player]
```

Let's walk through it in the same way as before: If the list with random numbers is empty or the iterationcounter $i \leq 0$ just return the current state of the population. Otherwise, we will have to run random numbers of iterations with a maximum count of `maxIter` calculate the new population distribution, generate the new population, add the result to a history for inspection and move on to the next tournament. That is essentially the workflow of the assignments in the `where` part.

1. `iteration` calculates the updated population with the `runIteration` method we saw earlier

2. `iterStats`[10] calculates summed up payout for each type in `playerTypes`. This is achieved by filtering the updated population by every type and then summing up the payments of the player instances histories. Because `iteration :: [Player]` we map the function `sum` $\circ$ `getPayments` over the updated population and sum up the results[11].

[10]short for iteration statistics
[11]`getPayments` first extracts a players history and then retrieves its payments into a list.

3. `payments` (the overall payout) is then calculated as the sum of the individual types' payments.

4. `newPopulationStats` calculates the new count of each player types' instances in the new population using the method `calcCount`[12]

5. `newPopulation` then calculates a new population based on the calculated distribution.

This function concludes the game logic as it is now possible to run tournaments.

5 Game Logic Wrapper

Becuse the type of the `runGame` function is rather complex and because we need a list of random integers to run the simulation, we defined a wrapper function that should make the code straigt forward to use:

```
startSimulation :: Int -> Int -> Int -> IO ([[(Int->Player, Int)]], Population)
startSimulation genSize tournaments iterations = do
    g <- getStdGen
    let gen = generatePopulation $
                map (\p-> (p, genSize `div` (length playerTypes))) playerTypes
        randList = randoms g
    putStrLn "Simulating Iterated prisoner"
    putStrLn $ "Population " ++ show (stats gen)
    return $ runGame tournaments iterations ([], gen) randList
```

This function first retrieves a (pseudo-)random number generator, generates a population with a given size, constructs an infinite list of random integers, prints the statistics for the population for convenience and then runs the game with the constructed data. Users can now run the code as discribed in the `main` function. Probably the best way to use the project is not to compile the code but to load it into ghci, play with the parameters and inspect the results. For large problem sizes[13] the program can be compiled with the `-threaded` flag and then run with +`RTS -Nx` where x = number of cores.

[12]`calcCount` and some other helper functions are listed in the appendix.
[13]large being defined by the specs of the computer the simulation is run on

6 Appendix

The code in this section is helper code that is rather self explainatory.

```
shuffle :: RandList -> [a] -> [a]
shuffle rands xs = let
  ys = take (length xs) rands
  in
  map fst $ sortBy (compare `on` snd) (zip xs ys)

makePairs :: [a] -> [(a,a)]
makePairs []       = []
makePairs [_]      = []
makePairs (h:h':t) = (h,h'):(makePairs t)

undoPairs :: [(a,a)] -> [a]
undoPairs [] = []
undoPairs ((a,b):t) = [a,b]++(undoPairs t)

stats :: Population -> [(String, Int)]
stats l = map (\p -> (name p, length $ filter (\e->name e == name p) l)) $
                 nubBy (\p1 p2 -> name p1 == name p2) l

-- tries to preserve the calculated amount for each player as close as possible
--          player payout       overall payout    population size
calcCount :: Int ->             Int ->            Int                 -> Int
calcCount _ 0 _ = 0
calcCount _ _ 0 = 0
calcCount a g p = let a' = fromIntegral a
                      g' = fromIntegral g
                      p' = fromIntegral p
                      in round $ a'/g'*p'
```