KI für allgemeine Strategiespiele

Sven Eric Panitz

14. Mai 2021

Inhaltsverzeichnis

1 Strategiespiele

In diesem Übungsblatt sei eine Typklasse für Strategiespiele für zwei abwechselnd spielende Spieler gegeben. Es gibt keine Zufallskomponente in diesen Spielen.

Als eine Besonderheit verwenden wir eine nicht Standarderweiterung von Haskell. Diese erlaubt es, eine Typklassen mit mehreren Typparametern zu definieren. Die entsprechende Erweiterung kann im Quelltext als Direktive angegeben werden.

Haskell: StrategicGames

```
> {-# LANGUAGE MultiParamTypeClasses #-}
>
> module StrategicGames where
```

Aus der Listenbibliothek benötigen wir die Maximumfunktion:

Haskell: StrategicGames

```
> import Data.List(maximumBy,transpose)
> import Data.List.Lens
> import Control.Lens.Combinators
> import Control.Lens.Operators
> import Control.Parallel.Strategies
> import Data.List.Utils
> import Control.Exception
> import Control.Monad
```

Es gibt in unseren Spielen Spielfelder, die von je einem der Spieler belegt sind, oder noch frei sind. Dieses drückt der folgende Aufzählungstyp aus:

Haskell: StrategicGames

```
> data Player = One  |Two  | None deriving (Eq)

> nextPlayer One = Two
> nextPlayer Two = One

> instance Show Player where
>  show One = "X"
>  show Two = "O"
>  show _   = " "
```

1.1 Die Typklasse `Game`

Wir schreiben eine allegemeine Typklasse, mit er wir beliebige zwei Spieler Strategiespiele beschreiben können. Die Typklasse vereinigt zwei Typen. Einen Typ für das eigentliche Spiel `g` und einen Typen, der Spielzüge beschreibt `m`.

Spiele sollen, damit man Sie mit einem einfachen Kommandozeilenprogramm spielen kann, auch immer einer Instanz der Klasse `Show` sein. Es ist zu empfehlen ein optisch guten String bei `show` zu erzeugen.

Haskell: StrategicGames

```
> class Show m => Game g m where
```

Der Typ `g` wird dabei den kind `* -> *` haben. Sein Argumenttyp wird wiederum der Typ `m` sein.

1.1.1 Abstrakte Funktionen

Um ein Strategiespiel für die Klasse `Game` zu implementieren, sind minimal 5 Funktionen zu schreiben.

Die erste Funktion wird benötigt, um einen Startzustand des Spiels zu erzeugen:

Haskell: StrategicGames

```
>   start :: (g m)
```

Wie man hier schön sieht, ist `m` der Argumenttyp des Spiels. Also ein Spiel `g` das Spielzüge `m` hat.

Desweiteren muss ein Spielzustand sagen können, welcher Spieler jetzt am Zug ist.

Haskell: StrategicGames

```
>   currentPlayer :: (g m)  -> Player
```

Alle für den aktuellen Spielzustand möglichen Spielzüge sind anzugeben. Der String gibt eine lesbare Version des Spielzuges an. Bei Spielen wie zum Beispiel Mühle kann dieses ja eine recht komplexe Information sein.

Haskell: StrategicGames

```
>   moves ::  (g m)  -> [m]
```

Eine Funktion wird benötigt, um auf einem Spielzustand einen bestimmten Zug auszuführen, um den Nachfolgezustand zu erhalten.

Haskell: StrategicGames

```
>   makeMove :: g m -> m -> g m
```

Und schließlich wird eine Funktion benötigt, die angibt, ob in diesem Zustand ein Spieler gewonnen hat.

Haskell: StrategicGames

```
>   hasWinner :: g m -> Bool
```

1.1.2 Funktionen mit Default-Implementierungen

Wenn man die obigen fünf abstrakten Funktionen implementiert, kann man für einfache Spiele wie Tic-Tac-Toe bereits eine voll spielfähige Variante mit Künstlicher Intelligenz erhalten. Bei komplexeren Spielen sind einiger der nun folgenden Default-Funktionen allerdings zu überschreiben.

Ein Spiel ist logischer Weise zu Ende, wenn es keine Spielzüge mehr gibt, oder ein Spieler gewonnen hat. (Letzteres könnte moves aber auch schon berücksichtigen.)

Haskell: StrategicGames

```
>   gameOver :: g m -> Bool
>   gameOver g  = null (moves g)||hasWinner g
```

Eine wichtige Information ist, welchen Wert der Spielzustand für den Spieler hat, der als nächstes dran mit Ziehen ist. Bei einfachen Spielen reicht es, nur einen Minimalwert zu geben, wenn es einen Sieger gibt, denn dann hat der Spieler, der jetzt an der Reihe wäre, verloren.

Bei komplexeren Spielen sind hier aber komplexere Bewertungen vorzunehmen, weil die Komplexität zu hoch ist, das Spiel so weit durchzuspielen, bis ein Spieler gewonnen hat. Schon bei „4-Gewinnt“ sind hier feinere Maße, wie die Anzahl der Tupel und Tripel anzugeben.

Haskell: StrategicGames

```
>   score :: g m -> Int
>   score g
>     |hasWinner g = -(winscore g)
>     |otherwise = 0
```

Auf jeden Fall, sollte ein Wert für die bestmöglichste Bewertung im Falle eines Sieges angegeben werden.

Haskell: StrategicGames

```
>   winscore :: g m -> Int
>   winscore _ = 1000000
```

Für die KI ist wichtig, wie tief im Spielbaum gesucht werden soll. Diese Information sollte das Spiel angeben können. Im Spielverlauf kann diese Information sogar variieren. So lange der Spielbaum noch weit in die Breite geht, kann die Suchtiefe nicht so hoch sein, wie wenn es pro Zug nur noch wenige Züge gibt.

Haskell: StrategicGames

```
>   searchDepth :: g m -> Int
>   searchDepth g = 10
```

Wir ermöglichen, dass die Funktion, die die Künstliche Intelligenz realisiert, für bestimmte Spiele unterschiedlich gesetzt werden kann. Standardmäßig wird die Alpha-Beta Suche für Spielbäume verwendet.

Haskell: StrategicGames

```
>   ai :: g m ->  AI (g m)
>   ai _ = alphaBeta
```

Für einen Spielzustand seien alle möglichen Spielzüge mit der KI bewertet. Hierzu wird der KI-Algorithmus auf alle Folgezustände angewendet.

Dieses ist die perfekte Stelle für eine Parallelisierung der Auswertung. Wir können parallel alle aktuell nöglichen Spielzüge mit der AI des Spiels bewerten.

Hierzu dient die Funktion `parMap`, die die aus dem Funktor bekannte Funktion `fmap` parallel auf die Elemente der Liste auswertet.

Haskell: StrategicGames

```
>   evalMoves :: g m -> [(m,Int)]
>   evalMoves g = parMap
>    (evalTuple2 r0 rdeepseq)
>    (\m-> (m, (ai g) (searchDepth g) (makeMove g m)))
>    (moves g)
```

Der beste Zug ist natürlich der, mit der besten Bewertung durch die KI.

Haskell: StrategicGames

```
>   bestMove ::  g m -> m
>   bestMove g
>     = fst
>         $maximumBy (\(_,v1) (_,v2) ->if v1<v2 then LT else GT)
>         $evalMoves  g
```

Die KI für ein Spiel, bewertet für eine Suchtiefe einen Spielzustand.

Haskell: StrategicGames

```
> type  AI a = Int -> a -> Int
```

1.2 Künstliche Intelligenz

In diesem Abschnitt wird die KI allgemein für Spiele, die die Klasse Game implementieren, entwickelt.

1.2.1 Spielbaum

Hierzu benötigen wir eine Baumstruktur für die Spielbäume:

Haskell: StrategicGames

```
> data Tree a = Node a [Tree a]  deriving Show
```

Die folgende Funktion erzeugt den kompletten Spielbaum für ein Spiel. Da Haskell nichtstrikt ausgewertet wird, wird immer nur der für die KI benötigte Teil dieses sicher sehr großen Baumes ausgewertet.

Haskell: StrategicGames

```
> createGameTree g =
>   Node g [createGameTree (makeMove g m)  |m <- moves g]
```

1.2.2 Min-Max Algorithmus

Der einfachste Algorithmus auf Spielbäumen ist der Min-Max-Algorithmus, der abwechselnd das MInimum und das Maximum aus den Ebenen auswählt.

Haskell: StrategicGames

```
> minimax depth g = -(minimax' depth $ createGameTree  g)
>  where
```

Gibt es keine Nachfolgespielzüge mehr, oder ist die Suchtiefe erreicht, so wird die Funktion score für die Bewertung herangezogen:

Haskell: StrategicGames

```
>   minimax' _ (Node g [])       = score g
>   minimax' 0 (Node g _)        = score g
```

Ansonsten wird abgestiegen in den Spielbaum und abwechselnd minimiert und maximiert, was sich durch die Negation der Ergebnisse erzielen lässt.

Haskell: StrategicGames

```
>   minimax' d (Node _ cs)
>      = maximum $ map (negate . (minimax' d)) $ cs
```

1.2.3 Alpha-Beta-Suche

Eine Optimierung des Min-Max-Algorithmus ist die Alpha-Beta-Suche, die hier unkommentiert folgt:

Haskell: StrategicGames

```
> alphaBeta depth g
>     = -alphaBeta' depth (-ws) (ws) (createGameTree g)
>  where
>    ws =  winscore g
>
>    alphaBeta' 0 _     _     (Node g _)  = score g
>    alphaBeta' _ _     _     (Node g []) = score g
>    alphaBeta' d alpha beta (Node _ cs) = falte (alpha,beta) crs
>      where
>        crs = map (\c-> -(alphaBeta' (d-1)(-beta)(-alpha) c)) cs
>
>        falte ab@(alpha,beta) (wert:werte)
>          |wert >= beta = beta
>          |wert > alpha = falte (wert,beta) werte
>          |otherwise    = falte ab werte
>        falte (alpha,_) [] = alpha
```

2 Spiel auf der Kommandozeile

Solange kein GUI für unser Spiel existiert, bietet sich eine kleine Kommandozeilenapplikation an.

2.1 Der Anwender ist am Zug

Wenn der Anwender am Zug ist, wird ihm der Spielzustand und die möglichen Spielzüge dargestellt. Er kann dann eine Spielzug eingeben. Bei komplexeren Datenstrukturen für Spielzüge kann er dieses mit Copy-Paste auf der Kommandozeile machen. Die Eingabe wird dann mit der Funktion `read` eingelesen.

Hierbei kann es zu einem Laufzeitfehler kommen, wenn die Eingabe sich nicht als Spielzug parsen lässt. Für diesen Fall bedienen wir und der Möglichkeit Ausnahmen abzufangen. Hierzu dient die Funktion `catch`.

Haskell: StrategicGames

```
> playUserMove a =
>   catch
>     (do
>        print a
>        putStrLn "Geben Sie jetzt Ihren Zug ein!"
>        putStrLn "Mögliche Züge:"
>        sequence $ map (putStrLn.show)$moves a
>        line <- getLine
>        let a1 = makeMove a $read line
>        print a1
>        return a1
>     )
>     (\e-> do
>       print (e :: SomeException)
>       playUserMove a
>     )
```

2.2 Der KI ist am Zug

Wenn die KI am Zug ist, so führt sie den bestmöglichen Zug aus.

Haskell: StrategicGames

```
> playAIMove a = do
>   putStrLn "Nun überlege ich meinen Zug.."
>   return$ makeMove a$ bestMove a
```

2.3 Die Spielschleife

Abwechselnd sind jetzt der Anwender und die KI am Zug. Das Spiel wird beendet, wenn einer gewonnen hat oder ein Remis festgestellt wurde.

Haskell: StrategicGames

```
> playGame a
>   |gameOver a = putStrLn "Das war ein Remis."
> playGame a = do
>   a1 <- playUserMove a
>   if (hasWinner a1) then putStrLn "Gratulation! Sie haben gewonnen."
>   else if not$gameOver a1
>   then do
>     a2 <- playAIMove  a1
>     if (hasWinner a2)
>     then do
>        print a2
>        putStrLn "Dieses Spiel hat der Rechner gewonnen."
>     else playGame a2
>   else playGame a1
```

3 Aufgabe: Tic Tac Toe

In dieser Aufgabe sollen Sie als erstes Spiel, das Spiel Tic-Tac-Toe entwickeln.

Hierzu sei folgende Datenstruktur gegeben:

Haskell: StrategicGames

```
> data TicTacToe a = TTT Player [Player]
```

Das Spielfeld wird durch eine Liste dargestellt. Diese soll immer die Länge 9 haben und die drei Reihen hintereinander geschrieben darstellen.

a) Machen Sie die Datenstruktur TicTacToe zu einer geeigneten Instanz von der Klasse Show.

Haskell: StrategicGames

```
> instance Show (TicTacToe m)  where
>   show (TTT npl board) = ""
```

b) Implementieren Sie nun die Klasse Game für den Datentyp TicTacToe.

Haskell: StrategicGames

```
> newtype P = P(Int, Int) deriving (Show,Read,Eq)

> instance Game TicTacToe P  where
```

Zunächst implementieren Sie die Funktion start:

Haskell: StrategicGames

```
>  start = TTT One [] --ToDo
```

c) Implementieren Sie die Funktion, die den aktuellen Spieler für den nächsten Zug angibt. Spieler One soll immer den ersten Zug haben.

Haskell: StrategicGames

```
>   currentPlayer ttt = None --todo
```

d) Implementieren Sie die Funktion moves, die alle möglichen Züge berechnet.

Haskell: StrategicGames

```
>   moves ttt =  [] --ToDo
```

e) Implementieren Sie die Funktion makeMove, die einen bestimmten Zug für ein Spiel ausführt.

Haskell: StrategicGames

```
>   makeMove ttt m = ttt --ToDo
```

f) Implementieren Sie die Funktion, die angibt ob ein Spieler gewonnen hat.

Haskell: StrategicGames

```
>   hasWinner ttt = False
```

Wenn Sie alles korrekt implementiert haben, können Sie jetzt gegen den Rechner spielen:

Haskell: StrategicGames

```
> playTicTacToe = playGame (start :: TicTacToe P)
```

4 Mühle

Als komplexeres Brettspiel soll mit unserer Rahmenbibliothek nun das Brettspiel ›Mühle‹[Wik19] umgesetzt werden. Mühle ist natürlich um einiges komplexer als TicTacToe und birgt einige Schwierigkeiten in der Umsetzung, die es komplexer machen als Spiele wie ›Vier Gewinnt‹ oder ›Reversi‹. Grund sind die drei unterschiedlichen Spielphasen. ZUsätzlich kann ein Zug unter Umständen nicht nur die Entscheidung über das Setzen des eigenen Steines, sondern die Wegnahme eines gegnerischen Steines beinhalten. Auch das Spielbrett ist nicht einfach ein zweidimensiolanes Gitter.

4.1 Datenstruktur

Für das Spielfeld bieten sich viele unterschiedliche Modellierungen als Datenstruktur an. Eine Möglicheit würen drei Listen mit jeweils acht Elementen, die die drei Quadrate des Spielfeldes darstellen. Wir haben uns für die quasi transponierte Darstellung entschieden, in der es acht Listen mit drei Elementen gibt. Diese drei Elemente sollen von außen nach innen die Felder an der selben Quadratposition in den drei Quadraten darstellen. Unterade Indizes in der Achterliste sind die Ecken, wobei wir mit Inex 0 beginnen.

Die Datenstruktur für den Spielzustand hält zusätzlich die Information über den Spieler, der als nächstes zieht. Weiterhin gibt es für jeden Spieler eine Zahl, die anzeigt, wie viele Steine er noch gar nicht ins Spiel gesetzt hat und eine Zahl, die für jeden Spieler anzeigt, wie viele Steine er aktuell auf dem Spielfeld hat.

Schließlich gibt es die Liste der in diesem Spielzustand möglichen Spielzüge.

Da es sich um eine komplexere Zusammenfassung von Informationen handelt, bauen wir nicht allein auf einen Konstruktor unter Verwendung von Pattern-Matching, sondern definieren einen Typ in der Record-Notation:

Haskell: StrategicGames

```
> data Muehle m = Brett
>   { brett::[[Player]]
>   , zuSetzen1::Int
>   , zuSetzen2::Int
>   , current::Player
>   , aufFeld1::Int
>   , aufFeld2::Int
>   , nextMoves::[MuehleZug]
>   }
```

Mühle kennt drei Spielphasen:

- Setzphase: Spieler setzen Spielsteine auf freie Felder.
- Zugphase: Spieler schieben Steine auf ein freies, verbundenes Nachbarfeld.

- Sprungphase: Ein Spieler hat nur noch drei Steine und kann auf ein beliebiges freies Feld springen.

Wir benötigen trotzdem nur zwei unterschiedliche Züge: einen zum Setzen und einen zum Umsetzen. Schieben und Springen sind beides Umsetzzüge, die von einem Feld auf ein anderes umsetzen.

Ein Zug kann beinhalten, dass er eine Mühle erzeugt und somit ein Spielstein des Gegners vom Spielfeld genommen werden kann. Hierfür nutzen wir den Typ `Maybe`.

Die Felder des Spiels sind über die Listenindizes indiziert. Der erste Index gibt an, auf welcher Position des Quadrates das Feld liegt, der zweite Index gibt an, auf welchem der drei Quadrate das Feld liegt.

Haskell: StrategicGames

```
> data MuehleZug = Setze (Int,Int) (Maybe (Int,Int))
>  |Schiebe (Int,Int)(Int,Int) (Maybe (Int,Int))
>    deriving (Show,Eq,Read)
```

Wichtig sind die Read- und Show-Instanzen, die benötigt werden, um das Spiel auf der Kommandozeile zu spielen.

4.2 Darstellung als String

In diesem Lehrbrief verwenden wir kein GUI sondern bieten nur eine Möglichkeit über die Kommandozeile an, ein Spiel zu spielen. Somit kommt der Stringdarstellung des Spielfeldes natürlich eine wichtige Rolle zu.

Für die Umsetzung der Funktion `show` sei eine Schablone des Spielbrettes als String gezeichnet, in der die 24 Felder mit Buchstaben bezeichnet sind. Diese Buchstaben werden durch ein Aufruf von `replace` mit dem im aktuellen Spielstand dort platzierten Spieler ersetzt.

Zusätzlich werden die noch nicht auf das Spielbrett gesetzten Steine unterhalb des Spielfeldes angezeigt.

Haskell: StrategicGames

```
> instance Show (Muehle m)  where
>  show (Brett{brett=brett, zuSetzen1=spieler1, zuSetzen2=spieler2}) =
>       replaceall replacements template
>    ++ take spieler1 (repeat 'X') ++ "\n"
>    ++ take spieler2 (repeat 'O') ++ "\n"
>   where
>    template
>      ="v----------a----------d\n"++
>       "|          |          |\n"++
>       "|   w------b------e   |\n"++
>       "|   |      |      |   |\n"++
>       "|   |   x--c--f   |   |\n"++
>       "|   |   |     |   |   |\n"++
>       "s---t---u     i---h---g\n"++
>       "|   |   |     |   |   |\n"++
>       "|   |   r--o--l   |   |\n"++
>       "|   |      |      |   |\n"++
>       "|   q------n------k   |\n"++
>       "|          |          |\n"++
>       "p----------m----------j\n"
>
>    replacements =  map (\(c,pl)-> replace [c] (show pl))
>                    $zip ['a'..] (concat brett)
>    replaceall rplfs str = foldl (\b f -> f b) str rplfs
```

4.3 Hilfsfunktionen

Zur Behandlung der Datenstruktur benötigen wir einige Hilfsfunktionen, um die Spiellogik etwas bequemer fomulieren zu können.

Setzen von Steinen Als erstes sei eine Funktion definiert, die auf einem Spielbrett einen Spieler auf eine bestimmte Position setzt.

Wir verwenden hierzu eine sogenannte Optik aus den Modulen:
`Data.List.Lens`, `Control.Lens.Combinators` und `Control.Lens.Operators`.

Haskell: StrategicGames

```
> setxy (x,y) brett pl = brett & element x .~ (brett!!x & element y .~pl)
```

Indiziertes Spielfeld Manchmal brauchen wir das Spielfeld mit den passendes Indizes und nicht nur mit der Spielerinformation.

Haskell: StrategicGames

```
> mkIndex brett = map (\(x,rs)-> map (\(y,p)->((x,y),p))$zip [0,1..] rs)
>                 $zip [0,1..] brett
```

Verbundene Nachbarn Für eine Feld ist es manchmal wichtig, die Nachbarfelder, die mit einer Linie verbunden sind, zu erfragen. Das ist eigentlich nur sinnvoll, wenn das Spielfeld indiziert wurde.

Haskell: StrategicGames

```
> getNeighbours (x,y) br
>   |mod x 2==0 && y==1
>     =  [br !!mod (x+1)8!!y,br !!mod (x-1) 8!!y,br !!x!!0,br !!x!!2]
>   |mod x 2==0
>     =  [br !!mod (x+1)8!!y,br !!mod (x-1) 8!!y,br !!x!!1]
>   |otherwise
>     =  [br !!mod (x+1)8!!y,br !!mod (x-1) 8!!y]
```

Check auf Mühle Ein wichtiger Check ist natürlich, ob ein Spielfeld in einer Mühle ist.

Haskell: StrategicGames

```
> inMuehle (x,y) br
>   |p==None = False
>   |mod x 2==0
>     =    all ((==)r) rs
>       || br!!(mod (x-1) 8)!!y==p && br!!(mod (x+1) 8)!!y==p
>   |otherwise
>     =   br!!(mod (x+1) 8)!!y==p && br!!(mod (x+2) 8)!!y==p
>       ||br!!(mod (x-1) 8)!!y==p && br!!(mod (x-2) 8)!!y==p
>     where
>       xs@(r:rs) = br!!x
>       p = xs!!y
```

Alle Mühlentriplets Auch kann es interessant sein, alle möglichen Mühlepositionen des Spielbrettes zu betrachten.

Haskell: StrategicGames

```
> nLets n [] = []
> nLets n xs = take n xs:nLets n (drop (n-1) xs)
```

Haskell: StrategicGames

```
> everySnd [] = []
> everySnd [x] = [x]
> everySnd (x:_:xs) = x:everySnd xs
```

Haskell: StrategicGames

```
> muehlen br = (everySnd br)
>   ++ (concat$map ((nLets 3).(\xs->last xs:xs)) (transpose br))
```

Einfügen der Züge Die Datenstruktur `Muehle` enthält eine Liste von den aktuell möglichen Zügen. Diese werden wir in der Funktion `muehleMoves` berechnen. Damit können sie dann in die Datenstruktur eingefügt werden.

Haskell: StrategicGames

```
> calcMoves b = b{nextMoves=muehleMoves b}
```

4.4 Implementierung der Game Typklasse

Nun können wir die Typklasse `Game` mit den Typen `Muehle` und `Muehlezug` implementieren.

Haskell: StrategicGames

```
> instance Game Muehle MuehleZug  where
```

4.4.1 Spielinitialisierung

Der Anfangszustand sind ein nur mit `None` besetztes Spielfeld, jeder Spieler hat neun Steine zum Setzen, keiunen Stein auf dem Spielfeld und der erste Spieler beginnt. Mit `calcMoves` lassen wir die ersten Spielzüge initialisieren.

Haskell: StrategicGames

```
>   start =
>    calcMoves$Brett (take 8$repeat [None,None,None]) 9 9 One 0 0 []
```

4.4.2 Der Spieler am Zug

Der aktuelle Spieler ist direkt im Datentyp vermerkt.

Haskell: StrategicGames

```
>   currentPlayer = current
```

4.4.3 Spielzüge

Auch die gerade möglichen Spielzüge sind vermerkt.

Haskell: StrategicGames

```
>   moves g = nextMoves g
```

4.4.4 Ausführung eines Spielzuges

Zum Setzen eines Spielzuges, sind die beiden Spielzugarten zu unterscheiden.

Setzphase Beim Setzen sind beliebig freie Felder als Ziel des Zuges möglich. Da viele Informationen sich auf den Spieler, der am Zug ist, beziehen, unterscheiden wir nach dem aktuellen Spieler:

Haskell: StrategicGames

```
>   makeMove g@(Brett{brett=br,zuSetzen1=p1,zuSetzen2=p2,current=cpl})
>            (Setze (x,y) rm)
>     = calcMoves$
>        if cpl==One
>         then Brett nb (p1-1) p2 np
>           (1+aufFeld1 g) (maybe (aufFeld2 g)(\_->(aufFeld2 g)-1) rm) []
>         else Brett nb p1 (p2-1) np
>           (maybe (aufFeld1 g)(\_->(aufFeld1 g)-1) rm) (1+aufFeld2 g) []
>       where
>        gesetzt = setxy (x,y) br cpl
>        nb = maybe gesetzt (\xy -> (setxy xy gesetzt None)) rm
>        np = nextPlayer cpl
```

Schiebe-/Sprungphase Analog geht es beim Schieben. Hier ist zusätzlich das Startfeld des Zuges zu leeren:

Haskell: StrategicGames

```
>   makeMove g@(Brett{brett=br,zuSetzen1=p1,zuSetzen2=p2,current=cpl})
>            (Schiebe (x,y) (u,v) rm)
>     = calcMoves
>         $Brett nb 0 0 (nextPlayer cpl)
>             (if cpl==One
>              then (aufFeld1 g)
>              else (maybe (aufFeld1 g) (\_->(aufFeld1 g)-1)rm ) )
>             (if cpl==Two
>              then (aufFeld2 g)
>              else (maybe (aufFeld2 g) (\_->(aufFeld2 g)-1)rm ) )
>             []
>       where
>        gesetzt = setxy (u,v) (setxy (x,y) br None) cpl
>        nb = maybe gesetzt (\xy -> (setxy xy gesetzt None)) rm
```

4.4.5 Ermittlung eines Siegers

Ein Sieger ist schnell ermittelt. In der Setzphase gibt es noch keinen Sieger, ansonsten ist gibt es einen Sieger, wenn ein Spieler nur noch zwei Spielsteine auf dem Brett hat.

Haskell: StrategicGames

```
>   hasWinner g = null$moves g
```

4.4.6 Spielbewertung

Haskell: StrategicGames

```
>   score g@(Brett{brett=br,current=cu})
>     |hasWinner g = -(winscore g)
>     |otherwise = 1000*currents-1000*others
>                  + calcPairs cu - calcPairs (nextPlayer cu)
>    where
>      brc = concat br
>      mue = muehlen br
>      others   = length$filter (== (nextPlayer cu)) brc
>      currents = length$filter (== cu) brc
>      pair p xs =    (length$filter (==p) xs)==2
>                  && (length$filter (==None) xs)==1
>      calcPairs p = 10*(length$filter (pair p) mue)
```

4.4.7 Suchtiefe

Eine vernünftige Suchtiefe für das Mühlespiel zu definieren ist gar nicht so einfach. Die Anzahl der möglichen Züge kann in den unterschiedlichen Spielphasen stark variieren. Anfangs stehen 24 Felder zum Setzen zur Verfügung. In der Schiebephase gibt es manchmal nur sehr wenige Spielzüge, in der Sprungphase wiederum sind viele Felder frei und als Ziel anzuspringen. Somit ist die Suchtiefe in Abhängigkeit von den Spielphasen unterschiedliche zu setzen. Wir variieren die Suchtiefe zwischen 4 und 10 Zügen.

Haskell: StrategicGames

```
>   searchDepth Brett{zuSetzen1=0,zuSetzen2=0,aufFeld1=a1,aufFeld2=a2}
>     |a1<=4 && a2<=4 = 4
>     |a1<=4 || a2<=4 = 4
>   searchDepth Brett{zuSetzen1=p1,zuSetzen2=p2}
>     |p1+p2 < 6 = 6
>     |p1+p2 < 4 = 8
>     |p1+p2 < 2 = 10
>   searchDepth _ = 4
```

4.4.8 Zugberechnung

Die Berechnung der Spielzüge ist in Mühle relativ komplex. Schuld daran sind primär die unterschiedlichen Spielphasen.

Haskell: StrategicGames

```
> muehleMoves g@(Brett{brett=br,zuSetzen1=p1,zuSetzen2=p2,current=cpl})
```

Wir unterscheiden die Phasen des Spiels:

Die erste Phase ist das Spielende, weil ein Spieler gewonnen hat.

Haskell: StrategicGames

```
>     |p1==0&&3 > aufFeld1 g  || p2==0&&3 > aufFeld2 g  = []
```

Solange ein Spieler noch Spielsteine nicht gesetzt hat, sind wir in der Setzphase

Haskell: StrategicGames

```
>     |p1+p2>0   = setzen
```

Wenn der aktuelle Spieler genau drei Steine auf dem Spielfeld hat, dann ist er in der Sprungphase.

Haskell: StrategicGames

```
>     |3==if cpl == One then aufFeld1 g else aufFeld2 g = springen
```

Ansonsten sind wir in der klassischen Schiebephase.

Haskell: StrategicGames

```
>     |otherwise = schieben
```

Schiebezug Für einen Schiebezug induzieren wir zunächst das Spielfeld.

Haskell: StrategicGames

```
>       where
>        indexed   = mkIndex br
>        coindexed = concat indexed
```

Nun filtern wir alle Positionen, auf denen der aktuelle Spieler sitzt. Dann betrachten wir dessen Nachbarn und filtern diese nach leeren Feldern. Das Ergebnis ist `myIndex`, eine Liste von Start und Zielpositionen eines Zuges.

Haskell: StrategicGames

```
>        myIndex = map (\(from,(to,_)) -> (from,to))
>          $filter (\(_,(_,pl))-> pl==None)
>          $concat
>          $map (\(pos,_)->(map (\x->(pos,x))$ getNeighbours pos indexed))
>          $filter (\(pos,p)->p==cpl)
>          $coindexed
```

Jetzt ist für jeden dieser Züge zu betrachten, ob durch ihn eine Mühle entsteht und welche Steine des Gegners anschließend weggenommen werden können.

Haskell: StrategicGames

```
>       schieben = concat$map (mkRvs.mkSchiebe) myIndex
>       mkSchiebe (from,to) = Schiebe from to Nothing

>       mkRvs s
>        |inMuehle to newBrett = [setRm (Just rm) s|(rm,_)<-removeables]
>        |otherwise = [s]
>         where
>           to = getTo s
>
>           Brett{brett=newBrett,current=newCurrent} = makeMove g s
>           newIndexed  = concat$mkIndex newBrett
>           removeables
>            = filter(\(pos,p)-> p==newCurrent
>                                &&not (inMuehle pos newBrett)) newIndexed
```

Dabei brauchen wir für beide Arten von Zügen eine Funtion, zum Setzen der Wegnahme eines gegerischen Steins und eine Funktion um das Ziel des Zuges zu ermitteln:

Haskell: StrategicGames

```
>       getTo (Schiebe _ to _) = to
>       getTo (Setze to _) = to
>       setRm rm (Schiebe from to _) = Schiebe from to rm
>       setRm rm (Setze  to _) = Setze  to rm
```

Sprünge Die Sprünge sind sehr ähnlich zu den Schiebezügen. Das Ziel des Zuges ist nicht allein aus den Nachbarn zu ermitteln, sondern aus allen freien Felern.

Haskell: StrategicGames

```
>       springen = concat$map (mkRvs.mkSchiebe) mySprungIndex
>
>       mySprungIndex =  map (\(from,(to,_)) -> (from,to))
>         $concat
>         $map
>          (\(pos,_)->
>            (map (\x->(pos,x))$filter (\(_,pl)->pl==None)coindexed))
>         $filter (\(pos,p)->p==cpl)
>         $coindexed
```

Setzen Beim Setzen braucht kein Feld, von dem aus gezogen wird, ausgewählt werden.

Haskell: StrategicGames

```
>       setzen = concat$map (mkRvs.(\pos->Setze pos Nothing))
>                  $map fst$filter (\(_,p)->p==None)$concat$indexed
>
```

4.5 Spiel

Die Stunde der Wahrheit hat geschlagen. Können wir gegen die KI unserer Implementierung Mühle spielen? Wie stark ist unser Gegner. Wie schnell errechnet er seine Züge?

Hierzu kompilieren wir folgendes Programm:

hs: Muehle

```
module Main where
import StrategicGames
import GHC.IO.Encoding

main = do
  setLocaleEncoding utf8
  playGame (start::( Muehle MuehleZug))
```

Wir wollen die Parallelität auf mehreren Prozessorkernen nutzen. Hierzu ist der ghc mit dem Argument `-threaded` zu sterten und das Programm mit der Anzahl der zu verwendenden Prozessorkerne:

Shell:

```
anitz@px1:~$ ghc -O2 -threaded StrategicGames.lhs Muehle.hs
[1 of 2] Compiling StrategicGames   ( StrategicGames.lhs,
↪  StrategicGames.o )
[2 of 2] Compiling Main             ( Muehle.hs, Muehle.o )
Linking Muehle ...
panitz@px1:~$ ./Muehle +RTS -N8
 ---------- ----------
|          |          |
|    ------ ------    |
|   |      |      |   |
|   |    -- --    |   |
|   |   |     |   |   |
 --- ---       --- ---
|   |   |     |   |   |
|   |    -- --    |   |
|   |      |      |   |
|    ------ ------    |
|          |          |
 ---------- ----------
XXXXXXXXX
OOOOOOOOO

Geben Sie jetzt Ihren Zug ein!
Mögliche Züge:
Setze (0,0) Nothing
Setze (0,1) Nothing
....
```

5 Lernzuwachs

Sehr viele Konzepte fanden in diesem Lehrbrief Anwendung:

- Abstraktion über Typklassen
- Multi-Parameter Typklasse
- Default-Funktionen in einer Typklasse
- potentiell unendliche Baumstruktur
- Abfangen einer Exception
- Recordsyntax
- Anwendung einer Optik
- Parallele Auswertung

Literatur

[Wik19] Wikipedia. Mühle (Spiel) — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 12-May-2021].