School of Electronic Engineering and Computer Science

MSc Software Engineering

Finding optimal strategies in sequential games with the novel selection monad

Johannes Niklas Hartmann
August 17, 2018

Hartmann, Johannes Niklas:
Finding optimal strategies in sequential games with the novel selection monad
Master Thesis Software Engineering
Queen Mary University London

Abstract

A potential bastract should be here!

Contents

List of Figures **iv**

1 Introduction **2**

1.1 Goal of the thesis 2

1.1.1 Case studies 3

1.1.2 Library 4

1.1.3 Performance 4

2 Introduction into Haskell **6**

2.1 Haskell is a Functional Programming Language 6

2.1.1 Type System 7

2.1.2 Functions 9

2.1.3 Modules 12

2.2 Monads 13

3 Selection Monad **15**

3.1 Selection Functions 15

3.1.1 Quantifier functions 17

3.1.2 Modelling sequential games 17

3.2 Implementing Games 19

3.2.1 Monadic selection function 20

3.2.2 Monadic bigotimes implementations 21

4 Library **22**

4.1 Structure . . . 22
4.2 Different min max functions . . . 23
4.2.1 Different ways of modelling the outcome . . . 25
4.2.2 Parallel versions . . . 27
4.2.3 Usage of this functions . . . 28

5 Example Game Implemenations 29
5.1 Connect Four . . . 29
5.1.1 Connect Three . . . 30
5.1.2 Interactive version . . . 33
5.1.3 Discussion . . . 34
5.2 Sudoku . . . 35
5.2.1 Implementation . . . 35
5.2.2 Discussion . . . 37
5.3 Simplified Chess . . . 38
5.3.1 Implementation . . . 38
5.3.2 Discussion . . . 41

6 Performance and Testing 42
6.1 Connect four . . . 43
6.1.1 Matrix instead of Lists . . . 43
6.1.2 Using parallel min max . . . 44
6.1.3 Discussion . . . 44
6.2 Sudoku . . . 45
6.2.1 Optimised version . . . 45
6.2.2 Parallel version . . . 46
6.2.3 Discussion . . . 46
6.3 Chess . . . 47
6.3.1 Runtime . . . 48
6.4 Discussion . . . 48
6.5 Testing of the library . . . 49

7 Discussion and Outlook **50**

7.1 Summary 50

7.2 Performance 51

7.3 Conclusion 51

7.4 Future work 52

Bibliography **52**

List of Figures

2.1 Example function definition 7
2.2 Pattern matching in the parameter definition 11
2.3 Pattern matching with the case statement 12
2.4 Example module definition 12
2.5 Example function for chaining two maybe operation 14
2.6 Definition of the monadic bind operator 14
2.7 Definition of the monad type class 14

3.1 Example selection function on strings 15
3.2 More general definition for a selection function 16
3.3 Definition for a maximum selection function 16
3.4 Type definition of the selection function type J 17
3.5 Type definition of the quantifier data type Basically the evaluation function is applied again to the result of the selection function. 17
3.6 Both have the same type but the first uses the previously defined custom data type J 18
3.7 *bigotimes* function definitions taken from [1, 2] 18
3.8 p type definition 19
3.9 epsilons type definition 19
3.10 Definitions calculating a perfect play 20
3.11 Selection monad definition taken from [1, 2] 20
3.12 *bigotimes* function definitions using the monadic behavour. Taken from [1, 2] 21

4.1 This three is representing a small example game where the red player wants to maximise the outcome and the black player wants to minimise it 24
4.2 Two type definitions of a potential maximum function. The second one uses the selection monad. Both type definitions express the same functionality. . 24
4.3 Generic minimum implementation 25
4.4 Three minimum implementation 26
4.5 Bool minimum implementation 26
4.6 Tuple minimum implementation 27
4.7 Parallel implementations of minimum functions 28
4.8 Example epsilons function using the tuple minimum and maximum function 28

5.1 Connect Three data types 31
5.2 Type signature of the Connect Three win function 31
5.3 Connect three utility functions 31
5.4 Connect three outcome function 32
5.5 Connect three epsilons function 33
5.6 Connect three main function 33
5.7 Connect three interactive epsilons function 34
5.8 Sudoku data type definition 36
5.9 Sudku: Different type signatures of the wins functions 36
5.10 Sudoku outcome function 36
5.11 Sudoku epsilons function 37
5.12 Chess data type definitions 39
5.13 Chess wins function 39
5.14 Functions that calculate all possible moves of the current player in chess. . 39
5.15 Chess outcome function 40
5.16 Chess epsilons function 40

6.1 Performance of the different Connect Three implementations 43
6.2 Performance of the different Connect Four implementations 44
6.3 Performance of the different Sudoku implementations 46
6.4 Performance of the simplified Chess implementations 48

Chapter 1

Introduction

The recently discovered monad, `Tx = Selection (x -> r) -> r`, can be used to elegantly find optimal strategies in sequential games. It can also be used to implement a computational version of the Tychonoff Theorem and to realize the Double-Negation Shift [1]. However, the aim of this masters project is to explore the monads ability to find optimal strategies in sequential games by implementing a collection of example games. This games can be seen as case studies that use the *selection monad* in the programming language Haskell. Additionally, common functionalities in this case studies were examined and summarised in a library.

1.1 Goal of the thesis

The goal of this thesis is to explore the ability of the *selection monad* to support elegant implementations of AI's playing sequential games. The approach of this AI hereby is to explore all possible moves for each player and to find a sequence of moves that lead to the optimal outcome. This project aims to explore the monads ability to do so, by implementing a collection of case studies. The feasibility of this monad to implement these case studies will be investigated in terms of performance of the implementation and

suitability of the selection monad for this case study. With the experience of implementing this case studies, a library was build. This library will enable future programmers an elegant and modular way of implementing AI's for sequential games.
Mainly the thesis consists of the following parts:

- Background research
- Library implementation
- 3 case study implementations
- Performance analysis of the case study
- Discussion of performance and suitability of the selection monad

1.1.1 Case studies

The case study games for this project are connect four, sudoku and a simplified version of chess. These games are all sequential games, which means a player has unlimited time to think about his move and eventually ends his turn by doing the move. This case studies are chosen to investigate different challenges that need to be solved in the games implementation. Connect four is a classic example of a two player game with three possible outcomes for each player (Win, Draw, Loose). Two versions of connect four where implemented, a simplified version where a human player plays against an AI and a version that computes the optimal winning strategy. Sudoku on the other hand is a one player game with only two possible outcomes (Valid and Invalid board). The simplified version of chess concentrates on solving chess endgame's with limited pieces (only Queen, King, Rook and Bishop). Hereby the challenge is to show that even a sophisticated game like chess can be implemented to some extend with the help of the *selection monad*. The case studies will be introduced in more detail in Section 5.

1.1.2 Library

As a result of the case studies, a library was build that contains all *selection monad* specific implementations as well as some helper functions that can be used to build the AI for a sequential game. The goal of this library is to summarise all common functionalities of the case studies in order to avoid code duplicates and also enable a modular programming approach when implementing sequential games. The library contains the definition of the *selection monad* with the related functions, a set of minimum and maximum functions used to describe the behaviour of the AI and functions that compute the next move of the AI as well as optimal plays and the corresponding optimal outcome. The design of the library will be explained in detail in Section 4.

1.1.3 Performance

In order to compute an optimal play for a sequential game, a computer programme needs to explore all possible games. As this can be a lot, for example connect four has 4,531,985,219,092 different possibilities [3], it is important to take performance optimisations into account when implementing this game.Some optimisations are already applied to the min or max functions of the library. It is also important to implement efficient functions that calculate the outcome of a game as well as calculating all possible moves. Due to the high number of possible moves, it is often impossible to explore all possibilities. In this case the AI must be limited to search only a certain number of moves in advance before making a choice. In Section 7.2 the performance of the example case studies will be examined and performance optimisations to their implementation will be explained as well as the performance optimisations of the library will be explained in detail.

This thesis starts in Chapter 2 with a basic introduction into the programming language Haskell for reader that are not familiar with Haskell. Chapter 3 will explain the concept of the novel selection monad and explains how this monad can be used to implement AI's for sequential games. Chapter 4 is then introducing the library that was made and Chapter 5 is then introducing three example games that are used as case studies. The performance of this case studies is then discussed further in Chapter 7.2. The paper concludes with the discussion and conclusion is in Chapter 7.

Chapter 2

Introduction into Haskell

The example case studies of this project are implemented in the programming language Haskell. Haskell is a pure functional programming language, which enables an elegant implementation of the novel *selection monad* and its functionality as well as well structured, declarative and modular example implementations of sequential two player games. The following Chapter is a brief introduction to the basic concepts of Haskell to support the understanding of the following Chapters. Section 2.2 is introducing the concept of the Monad data type and it's implementation in Haskell.

2.1 Haskell is a Functional Programming Language

As a pure functional programming language, Haskell enables modular function definitions. A Haskell program consists of small function definitions that are combined to bigger functions, which eventually form a whole program. A function in Haskell consist of an arbitrary number of input parameter and computes one output. The definition of "pure" in "pure functional programming language" forbids the presence of state/side effects and therefore functions in Haskell cannot have side effects. The result of a function therefore only depends on it's input parameters. Side effects and state depended computations are

modelled with the help of Monads which will be explained further in Section 2.2.
Haskell is also a strongly typed language, having a strong differentiation between the data types. Every function has a specific data type describing the input parameters and the output.

2.1.1 Type System

Haskell supports the commonly used data types: Integer (Int), Floating-point (Float), Boolean (Bool), String (String), Character (Char) ect. The type of a function will be annotated above the actual function definition with the following syntax:

```
functionName :: Int -> Int -> Bool
functionName param1 param2 = <functionDefinition>
```

Figure 2.1: Example function definition

Type Variables

To enable more general functions that work for more then one specific type, Haskell has the concept of type variables. Instead of writing a specific type, type variables are used in the type definition of the function. The following example function type signature is of a function that works for any list of elements with type **a** and returns the length of the list:

```
length :: [a] -> Int
```

Higher Order Functions

Furthermore, Haskell also supports higher order functions. This class of functions are taking another function as parameter. A common example is the map function on lists, which applies a function to each element in a given list, returning a new list with the modified elements. The map function has the following type signature:

```
map :: (a -> b) -> [a] -> [b]
```

Custom Data Types

In addition to the build-in types, custom data types can be defined. The following definition defines a tree data type. An object of this data type is either a Leaf or a Node with two child trees.

```
data Tree = Node Tree Tree | Leaf
```

This custom data types can also contain already defined data types. We can therefore modify this definition so that each Node and Leaf also hold an Integer value:

```
data Tree = Node Int Tree Tree | Leaf Int
```

Furthermore, it also supports Type variables, resulting in a tree that can contain elements of every data type:

```
data Tree a = Node a Tree Tree | Leaf a
```

Type classes

Types can be grouped in type classes. A type class normally describes a common behaviour that all types in this type class share with each other. For example all types that have an equality relation belong to the **Eq** type class. For each type in this type class the equality function (==) is implemented and can be used. It is possible to restrict general functions with type variables to certain type classes. The following example is from a function that searches a list of **a**'s for a given element of type **a** and returns true if the list contains this element. To do so it needs to call the equality function (==). It therefore only works for types that are in the **Eq** type class.

```
contains :: (Eq a) => [a] -> a -> Bool
```

Other common type classes are Ord for types that can be ordered or Show for types that can be transformed to a string.

2.1.2 Functions

In Haskell functions are the main language feature. Even values are represented as nullary function, which is a function without input. A typical function definition consists of it's type definition and the actual function body.

```
functionName :: Type1 -> Type2 -> Type3
functionName param1 param2 = <functionDefinition>
```

The *functionDefinition* part will be replaced by an expression that evaluates to a value of the corresponding output type for the given input parameters. Haskell does not support imperative computations or loops. Instead of loops, recursion is used for repeating a computation several times. Haskell also does not differ between operators or functions

and therefore these operators are simple functions that are used infix.

The syntax for function applications is: first the function name and then its parameters. For example the add Function with the following type:

```
add :: Int -> Int -> Int
```

can be applied as the following, to perform 2 + 2:

```
four = add 2 2
```

This expression can also be written with the + function, the infix variant of the add function:

```
four = 2 + 2
```

Currying

The concept of currying describes a partial application of a function. It is possible in Haskell to apply a function only to a part of its parameters, resulting in a new function that is still waiting for the rest of the parameters before evaluating the result. For example a new function can be build that always adds one to a given Integer by partially applying the previously introduced add function to 1:

```
add1 :: Int -> Int
add1 = add 1
```

Lambda Expressions

Haskell supports the concept of lambda expressions to define an anonymous function inside an expression rather then defining and naming the function outside the expression.

The syntax of lambda expressions is the following:

```
(\x -> x + 1)
```

and can be used for example inside the application of the map function:

```
listPlus1 = map (\x -> x + 1) [1,2,3,4]
```

which is adding one to each element in the list resulting in the list *[2,3,4,5]* In contrast, the same definition without a lambda expression, using the previous defined add1 function, could look like this:

```
listPlus1 = map add1 [1,2,3,4]
```

Pattern Matching

Certain container data types, like lists or tuples as well as self defined data types like the previous defined tree data type, can be de-constructed using pattern matching. This can either be done directly at the function definition, for example with the Tree data type:

```
isLeaf :: Tree -> Bool
isLeaf (Node t1 t2) = False
isLeaf (Leaf)       = True
```

Figure 2.2: Pattern matching in the parameter definition

Here the deconstruction happens instead of naming the input parameter. Depending on the structure of the data type the corresponding expression will be evaluated. As an alternative the case keyword can be used inside an expression:

```
isLeaf :: Tree -> Bool
isLeaf t = case (t) of
     (Node t1 t2)  -> False
     (Leaf)        -> True
```

Figure 2.3: Pattern matching with the case statement

2.1.3 Modules

To organise a Haskell program, data types and collections of functions can be summarised into modules, which then can be imported into other Haskell programs. A module can be defined with the following syntax at the beginning of a file:

```
module ModuleName (function1, function2) where
   function1 :: Type1 -> Type2
   function1 = ...

   function2 :: Type2 -> Type3
   function2 = ...

   helperFunction :: Type1 -> Type3
   helperFunction = ...
```

Figure 2.4: Example module definition

The functions in the parentheses are the functions that will be exported and made available when importing this module.
On contrast, the *helperFunction* will not be made available. The module *Prelude* is the main Haskell module providing the basic data types like **Int** or **Bool** and basic functions like + or **map**.

2.2 Monads

Monad is special type class of the previous introduced type classes. The concept of a monad originates in the mathematical field of the category theory. There are many ways to explain a monad. The following section will only give a brief introduction. For more details, there are many good tutorials out there explaining monads [4]. Monads can be used in many different use-cases. The most popular one are modelling state-full behaviour or doing Input/Output actions. To understand the concept of monads consider the following example: We have a bunch of functions that are all taking one argument as input and produce an optional outcome:

```
function1 :: a -> Maybe b
```

The optional outcome will be modelled with the maybe data type which is either the result or nothing:

```
data Maybe a = Just a | Nothing
```

A possible example functions with this behaviour can be a function that is searching a list for a particular element and then returning this element if the list contains it or nothing if it is not in the list.
We want now to chain the functions so that the output of the previous function is the input of the next function. When doing this without the help of the monadic behaviour, we would need to unpack each output first and handle the *Nothing* case properly, resulting in really nested code. To avoid this the following operator would be helpful:

Now it is possible to chain two functions that produce a maybe. In order to make this function more general, the type can be reduced to:

```
(>>?) :: a -> (a -> Maybe b) -> (b -> Maybe c) -> Maybe c
(>>?) x f1 f2 = case f1 x of
                  Nothing -> Nothing
                  (Just a) -> f2 a
```

Figure 2.5: Example function for chaining two maybe operation

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) x f = case x of
              Nothing -> Nothing
              (Just a) -> f a
```

Figure 2.6: Definition of the monadic bind operator

Compared to the type class definition of monads the type signature is similar:

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

Figure 2.7: Definition of the monad type class

So in general, Monads are container data types that surround the actual data. If there are a bunch of functions all producing the same monadic output, all with different data types that are actually in the container, the monadic >>= function provides a way of chaining them without the need to manually unpack them.

Chapter 3

Selection Monad

The selection monad was first described in the context of finding optimal plays in sequential games by Martín Escardó and Paulo Oliva in 2010 [1]. This section is an introduction to the selection monad, and how it can be used to find optimal plays and strategies in sequential games.

3.1 Selection Functions

To understand the selection monad we first need to look at selection functions. That are functions that select an element out of a given list of elements based a internal evaluation on the elements. An example selection function could be a function that takes a list of strings as an input and is searching the whole list for a string with the length 3. If the list contains such a string, one of these strings will be the output, if not, a random string will be selected. An implementation of this function could be the following: This

```
selectionFunction :: [String] -> String
selectionFunction [x]  = x
selectionFunction (x:xs) = if length x == 3 then x else selectionFunction xs
```

Figure 3.1: Example selection function on strings

implementation is very specific and only works for strings and the selection function is embedded inside the function definition. To generalise this definition we can use type variables and higher order functions, to define a more general version that works for every type and for many selection behaviours. A more general implementation could be the following:

```
selectionFunction2 :: [a] -> (a -> Bool) -> a
selectionFunction2 [x] f = x
selectionFunction2 (x:xs) f = if f x then x else selectionFunction2 xs f
```

Figure 3.2: More general definition for a selection function

This implementation is still specific to selection functions that only work with booleans. To extend this to a function that works with a more general type of truth values r, we need to embedded parts of the selection behaviour in the function definition. For example we have a function that maps every value in the input list to an integer value and we want to select the value that is assigned the highest mapping. A possible implementation could look like this:

```
selectionFunctionMax :: (Ord b) => [a] -> (a -> b) -> a
selectionFunctionMax [] f = undefined
selectionFunctionMax xs f = last (sortOn f xs)
```

Figure 3.3: Definition for a maximum selection function

Last and *sortOn* are build in functions of Haskell prelude module. *SortOn* sorts a list with respect to a given function, while last returns the last element of a list. Note, this achieves a more general type definition with the cost of loosing generality in the function implementation itself.

With this in mind we can now define a new general type for the selection function. We do this using custom data types and type variables. Here the type variable r stands for a general type of truth values and x denotes the type variable of the list elements. The

type and the adjusted type signature of the maximum selection function can be defined as below:

```
type J r x = (x -> r) -> x

selectionFunctionMax2 :: (Ord b) => [a] -> J b a
```

Figure 3.4: Type definition of the selection function type J

3.1.1 Quantifier functions

Closely related to this new selection type is the quantifier type also described by Martín Escardó and Paulo Oliva in their paper from 2010 [1]. This quantifier type returns instead of the element we want to select, the result of the selection function. For example, in the context of the selection function that is looking for a string with the length of 3, the corresponding quantifier would return true if such a string exists and false if not. Because of this close relationship a overline function can be defined which converts the selection type to the quantifier type. The implementation could look like this:

```
type K r x = (x -> r) -> r

overline :: J r x -> K r x
overline e p = p(e p)
```

Figure 3.5: Type definition of the quantifier data type Basically the evaluation function is applied again to the result of the selection function.

3.1.2 Modelling sequential games

In order to bring this selection functions in the context of implementing an AI for sequential games, we can model the way the AI decides which move it wants to play as a selection function. This selection function is selecting the best possible move out of a list

of all possible moves. The type signature of this function could look like this:

```
bestMove :: [Move] -> J r Move
bestMove' :: [Move] -> (Move -> r) -> Move
```

Figure 3.6: Both have the same type but the first uses the previously defined custom data type J

where r is describing the current state of the game, for example if the player is winning or loosing after this move, and Move is an custom data type that is representing a move in an arbitrary game. Furthermore a whole play of a sequential game can be modelled as a list of all this *bestMove* selection functions:

```
sequentialPlay :: [[Move] -> J r Move]]
```

Martín Escardó and Paulo Oliva introduced in their paper from 2010 [1] the *bigotimes* function. This function is combining a list of selection functions into one big selection function. This combined selection function can be seen as the function selecting a list of optimal moves that represent a perfect play.

```
otimes :: J r x -> (x -> J r [x]) -> J r [x]
otimes e0 e1 p = a0 : a1
  where a0 = e0(\x0 -> overline (e1 x0) (\x1 -> p(x0:x1)))
        a1 = e1 a0 (\x1 -> p(a0 : x1))
        overline e p = p(e p)

bigotimes :: [[x] -> J r x] -> J r [x]
bigotimes [] = \p -> []
bigotimes (e : es) = e [] `otimes` (\x -> bigotimes[\xs->d(x:xs) | d <- es])
```

Figure 3.7: *bigotimes* function definitions taken from [1, 2]

The implementation of this bigotimes function as shown in Figure 3.7 is explained in detail in Martín Escardó and Paulo Oliva introduced in their paper from 2010 [1].

3.2 Implementing Games

With the help of this *bigotimes* function, it is possible to define functions that calculate an optimal play, the coutcome of an optimal play and the optimal strategy. When implementing a sequential game, the programmer needs to define two essential functions. First a function that is evaluating a given list of moves to a value that is representing the state of the game. Here often integer values are used where 1 stands for player 1 wins, 0 stands for a draw and -1 stands for player 2 wins. This function will be named p in the following case study implementations. The type signature of this function looks like this, where R is representing the state of the game:

```
p :: [Moves] -> R
```

Figure 3.8: p type definition

The second function is representing the strategies that both players use to play the game. It is a list of selection functions that select one move out of all possible moves. In the following case studies this function will be called *epsilons* and the type signature of this function is the following:

```
epsilons :: [[Move] -> J R Move]
```

Figure 3.9: epsilons type definition

Having this p and *epsilons* defined, the following three function can be called to to calculate the a optimal play:
In order to implement interactive games where the AI only chooses one move instead of the whole optimal play, the optimal strategy function calculates the whole optimal play but only returns the first move out of this optimal play list. It also has an additional parameter to take previous made moves into account.

```
optimalPlay :: ([b] -> a) -> [[b] -> J a b] -> [b]
optimalPlay p epsilons = bigotimes epsilons p

optimalOutcome :: ([b] -> a) -> [[b] -> J a b] -> a
optimalOutcome p epsilons = p $ optimalPlay p epsilons

optimalStrategy :: ([b] -> a) -> [[b] -> J a b] -> [b] -> b
optimalStrategy p epsilons as = head(bigotimes epsilons p')
   where p' xs = p(as ++ xs)
```

Figure 3.10: Definitions calculating a perfect play

3.2.1 Monadic selection function

This previously defined custom data type for selection functions can now be made instance of the monad type class.

Here should later be an explaination why it is helpful tu make J instance of monad and also explaining the following two code snippets.

```
newtype J r x = J {selection :: (x -> r) -> x}

instance Monad (J r) where
  return = unitJ
  e >>= f = muJ(functorJ f e)

morphismJK :: J r x -> K r x
morphismJK e = K(\p -> p(selection e p))

unitJ :: x -> J r x
unitJ x = J(\p -> x)

functorJ :: (x -> y) -> J r x -> J r y
functorJ f e = J(\q -> f(selection e (\x -> q(f x))))

muJ :: J r(J r x) -> J r x
muJ e = J(\p -> selection(selection e(\d -> quantifier(morphismJK d) p)) p)
```

Figure 3.11: Selection monad definition taken from [1, 2]

3.2.2 Monadic bigotimes implementations

Explaination how to use the monad functionality to implement bigotimes again.

```
otimes :: Monad m => m x -> (x -> m y) -> m(x, y)
xm `otimes` ym =
    do x <- xm
       y <- ym x
       return (x, y)

bigotimes :: Monad m => [[x] -> m x] -> m[x]
bigotimes [] = return []
bigotimes (xm : xms) =
    do x <- xm []
       xs <- bigotimes[\ys -> ym(x:ys) | ym <- xms]
       return(x : xs)
```

Figure 3.12: *bigotimes* function definitions using the monadic behavour. Taken from [1, 2]

Chapter 4

Library

The previously in Chapter 3 introduced functions that are calculating the optimal play as well as the monadic definition of the selection data type and the quantifier data type are summarised in a library. The goal of this library is to support the implementation of AI's of sequential games. The full implementation of the library can be found in the Appendix. The library was build by analysing previously existing case studies [2] and with the experience that was gained by implementing the example games introduced in Chapter 5. The example games where then edited to work with the library and therefore the library is introduced before the example game implementations. The following Chapter is explaining the library's structure and design, and also gives an overview on how to use the library to implement AI's for sequential games.

4.1 Structure

The library consists of five different sub modules, each containing a different aspect of the selection monad. The first two modules contain the definition and type class instances for the selection monad and the quantifier monad as defined in Figure 3.11. The next module is containing the *bigotimes* function definition as introduced in Figure 3.12. Another

module contains the optimal play functions from Figure 3.10. The final module provides some minimum and maximum functions that can be used to build the 3.9 function that where introduced in Figure 3.9. This minimum and maximum functions will be discussed further in Section 4.2.

In order to use the library a potential user needs to implement two functions, that where previously mentioned in Chapter 3: The *epsilons* and p function. Before implementing this two function, at least two data types need to be defined: The **Move** data type which represents a move in the game and the **R** data type representing the current state of the game. In an simple game the state of the game can be described as either winning, loosing or a draw, but the library also supports more sophisticated outcome types. Which types the library supports will be discussed in Section 4.2. Whit this data type definition, the p and *epsilons* function can now be defined. P is the function which is calculating the outcome of the game for a given list of previous moves. *Epsilons* is a list of selection functions, modelling the strategy which optimal players would use to make their move. Tu support the implementation of the *epsilons* function, the library provides a set of minimum and maximum selection functions that are already selection functions.

4.2 Different min max functions

The minimax algorithm is an algorithm in game theory that calculates optimal plays. With the help of the selection monad and the bigotimes function it is easy to define the optimal play functions as introduced in Figure 3.10. This is basically an implementation of this minimax algorithm. The algorithm explores all possible moves, and calculates the best move for each player at each stage. In this algorithm all possible moves of the game are represented as a tree. The algorithm is now exploring the whole tree bottom up and assigns each leaf a value. This value is representing the outcome of the game based on all previous played moves. Then each parent node will be assigned the value out of all child

node values that is the best for the current player. As one player tries to minimise the outcome and the other player wants to maximise the outcome minimum and maximum functions are used to find the optimal outcome. Figure 4.1 is showing an example tree of the minimax algorithm.

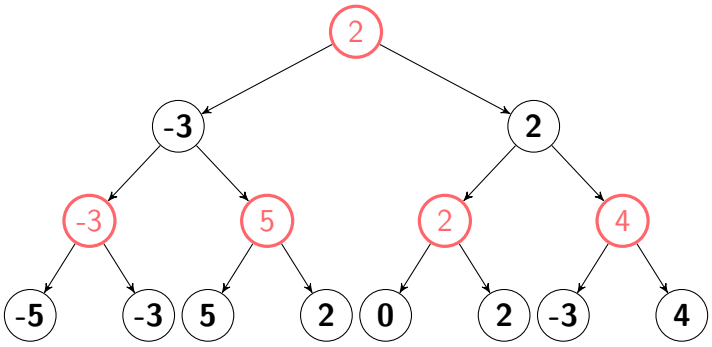

Figure 4.1: This three is representing a small example game where the red player wants to maximise the outcome and the black player wants to minimise it

The library is providing a set of different minimum and maximum function that are already using the selection monad and are suitable for different ways of outcome models. These functions select the value of a given list that is producing the highest outcome of a given outcome function. In the context of implementing games the type *a* is representing a move and **b** is representing the outcome of the game:

```
epsilonMax :: (Ord b) => [a] -> J b a
epsilonMax' :: (Ord b) => [a] -> (a -> b) -> a
```

Figure 4.2: Two type definitions of a potential maximum function. The second one uses the selection monad. Both type definitions express the same functionality.

4.2.1 Different ways of modelling the outcome

There are different ways of modelling the outcome of a game. Some require special minimum and maximum functions. Therefore and for performance reasons the library supports four different ways of modelling the outcome.

Generic

The easiest way of modelling the outcome is to use Integer or Float values. In this case a high value means that the first player is winning and a low value that the second player is winning. The implementation of the minimum and maximum functions is there a strait forward implementation:

```
epsilonMin :: (Ord b) => [a] -> J b a
epsilonMin xs = J(epsilonMin' xs)

epsilonMin' :: (Ord b) => [a] -> (a -> b) -> a
epsilonMin' [] _ = undefined
epsilonMin' xs f = head $ sortOn f xs
```

Figure 4.3: Generic minimum implementation

1,0,-1

Another way is still using integers but only use the numbers 1, 0 and -1. Here 1 means player one is winning, 0 that the game is a draw and -1 that player two is winning. To limit the upper and lower bound, it is possible to optimise the algorithm and stop searching when the maximum or minimum is found. The implementation of this algorithm looks like this:

```
epsilonMinThree :: [a] -> J Three a
epsilonMinThree xs = J(epsilonMin' xs)

epsilonMinThree' :: [a] -> (a -> Three) -> a
epsilonMinThree' [] _  = undefined
epsilonMinThree' [x] _ = x
epsilonMinThree' (x:xs) f | f x == 1 = epsilonMinThree' xs f
                          | f x == -1 = x
                          | otherwise = case findMin xs f of
                                        (Just y) -> y
                                        Nothing -> x
   where findMin [] _  = Nothing
         findMin (x:xs) f = if f x == -1 then Just x else findMin xs f
```

Figure 4.4: Three minimum implementation

Bool

Instead of three values it is also possible to just have just two values representing the outcome of the game. Here a boolean is used to represent the game outcome. The following implementation is also optimised to stop searching as soon as the minimum/maximum was found:

```
epsilonMinBool :: [a] -> J Bool a
epsilonMinBool xs = J(epsilonMinBool' xs)

epsilonMinBool' :: [a] -> (a -> Bool) -> a
epsilonMinBool' [] _  = undefined
epsilonMinBool' [x] _ = x
epsilonMinBool' (x:xs) f = if not $ f x then x else epsilonMinBool' xs f
```

Figure 4.5: Bool minimum implementation

Tuple

In many cases it is not enough to represent the outcome of the game, but it is also necessary to track after how many moves this outcome is reached. For example if an player is loosing

with every move he can make, he still wants to play as long as possible in case his opponent makes an mistake. This outcome type is often necessary in the case when implementing an interactive game where a real person plays against the computer. To model this outcome type a tuple is used consisting of two integers. The first is representing the outcome and the second how many moves it takes to achieve this outcome. The function is selecting the shortest way if it has a winning strategy otherwise it tries to stay as long as possible in the game. An implementation could look like this:

```
epsilonMinTuple :: [a] -> J (Int, Int) a
epsilonMinTuple xs = J(epsilonMinTuple' xs)

epsilonMinTuple' :: [a] -> (a -> (Int, Int)) -> a
epsilonMinTuple' [] _ = undefined
epsilonMinTuple' xs f = let list = sortOn fst (map (\x -> (f x, x)) xs) in
                        if  fst (fst $ head list) > 0
                        then snd $ last $ filter (\x -> fst (fst x) == fst
                            (fst $ head list)) list
                        else snd $ head list
```

Figure 4.6: Tuple minimum implementation

4.2.2 Parallel versions

For the generic and tuple version minimum and maximum function, it improves the performance of the whole algorithm when executing the outcome function in parallel. The implementation of this versions are always first applying the outcome function to each element before making a decision, where the other version stop searching when the maximum is found. This parallel versions have a huge impact on the performance and cut down the calculating time in some cases by the factor 100. The implementation is done with the build in Haskell parallel library. The Implementation of this parallel functions look like this:

```
epsilonMinParalell ::(NFData a, NFData b, Ord a, Ord b) => [b] -> J a b
epsilonMinParalell xs = J(epsilonMinParalell' xs)

epsilonMinParalell' :: (NFData a, NFData b, Ord a, Ord b) => [a] -> (a -> b)
    -> a
epsilonMinParalell' xs f = snd $ minimum $ parMap rdeepseq (\x -> (f x, x)) xs

epsilonMinTupleParalell :: (NFData a) => [a] -> J (Int, Int) a
epsilonMinTupleParalell xs = J(epsilonMinTupleParalell' xs)

epsilonMinTupleParalell' :: (NFData a) => [a] -> (a -> (Int, Int)) -> a
epsilonMinTupleParalell' [] _ = undefined
epsilonMinTupleParalell' xs f = let list = sortOn fst (parMap rdeepseq (\x ->
    (f x, x)) xs) in
                          if  fst (fst $ head list) > 0
                          then snd $ last $ filter (\x -> fst (fst x) == fst
                              (fst $ head list)) list
                          else snd $ head list
```

Figure 4.7: Parallel implementations of minimum functions

4.2.3 Usage of this functions

This predefined minimum and maximum functions can be used to define the *epsilons* function that is necessary for the optimal play computation. As introduced in Chapter 3 a game classical two player game is a list of decisions where each player decides its next move based on the previous moves that are done before. So a infinite list is build that alternate between minimum and maximum selection functions. In order to limit the search depth, only a certain number of elements are taken from the beginning of the list. An possible *epsilons* implementation that searches the tree of all possible moves to the depth of 9 and uses the previously introduced tuples outcome could look like this:

```
epsilons :: [[Move] -> J R Move]
epsilons = take 9 all
  where all = epsilon1 : epsilon2 : all
        epsilon1 history = epsilonMaxTupleParalell (getPossibleMoves history)
        epsilon2 history = epsilonMinTupleParalell (getPossibleMoves history)
```

Figure 4.8: Example epsilons function using the tuple minimum and maximum function

Chapter 5

Example Game Implemenations

This Section introduces the implementations of the three example games used as case studies to explore the selection monads ability to support AI implementations for sequential games. All case studies are implemented in the programming language Haskell. A brief introduction to Haskell can be found in Chapter 2. The following sections introduces the rules each of the games briefly. Then the implementation details of each game will be presented and finally the suitability of the selection monad will be discussed. Each of the case studies needed to deal with the complexity of the corresponding game. This complexity and the corresponding performance optimisations will be discussed separately Chapter 7.2.

5.1 Connect Four

Connect four is a sequential two player game. It is played on a 7x6 grid board. A turn consists of a player choosing a column of the board and inserting a disk of his colour. The disk is then falling down to the bottom of the board occupying the next available space in this column. The player who first has 4 in a row (Horizontal, Vertical, Diagonal) wins the game.

A perfect strategy to play this game was fist introduced by Victor Allis [5] in 1988. Allis performed a knowledge based approach, describing nine strategies leading to a win for the beginning player. A bruteforce approach, trying all of the the over four trillion different games [3], was done in 1995 by John Tromp [6]. In a perfect play, the beginning player can archive a win on the 41st move.

The following section introduces two different implementations of Connect Four, one that calculates the a perfect play and another interactive version, where a human player plays against an AI. A first implementation of the game with the selection monad showed that without any pruning (limiting of the search space of all possible moves), it is not feasible to calculate a perfect strategy for *connect four* because of the huge amount of different games that are needed to be explored. More details on the performance of this implementations are discussed in Chapter 7.2. In order to reduce the number of possible moves, the game was simplified to a connect three version where the board size was reduced to 5x3 (4x3 for the interactive version) and to win the game it is only necessary to have three in a row. For this simplified version it was possible to compute a perfect play in under a minute. The following section will introduce the implementation details.

5.1.1 Connect Three

In order to implement *connect three*, first four data types need to be defined. The board is represented as a matrix using a third party matrix library of Daniel Díaz [7]. This library enables constant access times and is used instead of nested lists in order to improve the performance. The board is then filled with values of type *Player* which are either Player 1 or Player 2 or Nothing (X,O,N). A move is represented as an Integer, indicating the column a disk is inserted into. The outcome R of a game is represented by a tuple of two Integers. The first integer represents the state of the game. **1** for player 1 is winning, **0** for a draw and **-1** for player 2 is winning. The second integer counts the number of moves

that are made before the current evaluation of the board.

```
type R = (Int,Int)
type Move = Int
type Board = Matrix Player
data Player = X | O | N
```

Figure 5.1: Connect Three data types

Next a *wins* function needs to be defined, evaluation for a given board and player whether a win was achieved. Therefore the whole board is checked for the winning criteria (Three in a row). The full implementation of this function can be found in the Supporting material.

```
wins :: Board -> Player -> Bool
```

Figure 5.2: Type signature of the Connect Three win function

Now utility functions are defined handling the insertion of a disk into the board and computing a list of possible moves depending on a list of previous moves. The *insert* function checks a column from the bottom and inserts the players disk at the first appearance of an empty cell (represented by an N). The *getPossibleMoves* function computes all remaining possible moves by checking how many discs were inserted into the corresponding column (It is only possible to insert 3 disks in a row).

```
insert :: Move -> Player -> Board -> Board
insert m = insert' (m,1)
  where insert' (x,y) p b = if b ! (x,y) == N
                            then setElem p (x, y) b
                            else insert' (x, y + 1) p b

getPossibleMoves :: [Move] -> [Move]
getPossibleMoves [] = [1..5]
getPossibleMoves xs = filter (\x -> length (elemIndices x xs) < 3) [1..5]
```

Figure 5.3: Connect three utility functions

Building the AI

In order to build the AI for the connect three game, the p and *epsilons* functions need to be defined. This functions will be introduced in Chapter 3 and explained in detail in Chapter 4. To define the p function that maps a list of previous played moves to the outcome, a helper outcome function is defined. This outcome function is taking the starting player, a list of played moves, a board and the count of moves that are already played on the given board, and inserts all moves from the list of moves into the given board. For each step the function checks if this move is a winning move. In this case it stops inserting and returns the outcome.

```
value :: Board -> Int
value b | wins b W = 1
        | wins b B = -1
        | otherwise = 0

outcome :: Player -> [Move] -> Board -> Int -> R
outcome _ [] b i     = (value b, i)
outcome p (m : ms) b i = let nb = insert m p b in
                           if wins nb p
                           then (value nb, i + 1)
                           else outcome (changePlayer p) ms nb (i + 1)

p :: [Move] -> R
p ms = outcome X ms (matrix 5 3 (const N)) 0
```

Figure 5.4: Connect three outcome function

To model the AI's behaviour the parallel minimum and maximum function are used. This functions are provided by the library. The first player wants to maximise the outcome and the second player wants to minimise the outcome. An infinite list of alternating Min and Max selection functions is build. By constantly increasing the search depth, a optimal play was found after 9 moves, and we therefore take only the first 9 elements of this infinite list of selection functions. The epsilon function was implemented like this:

```
epsilons :: [[Move] -> J R Move]
epsilons = take 9 all
  where all = epsilonX : epsilonO : all
        epsilonX h = epsilonMaxTupleParalell (getPossibleMoves h)
        epsilonO h = epsilonMinTupleParalell (getPossibleMoves h)
```

Figure 5.5: Connect three epsilons function

Then in the main function the optimal game function is called with the previous p and *epsilons* function.

```
main :: IO ()
main = do
  let optimalGame = optimalPlay p epsilons
  putStr ("An optimal play for " ++ gameName ++ " is "
     ++ show optimalGame
     ++ "\nand the optimal outcome is " ++ show (p optimalGame) ++ "\n")
```

Figure 5.6: Connect three main function

This is then calculating an optimal play where the first player wins after nine moves. The resulting play is: *[2,4,3,1,2,2,3,5,1]*.

5.1.2 Interactive version

In order to make an interactive version of this connect three game, the *epsilons* function needs a small adjustment. The new *epsilons* function needs to take the previous played moves into account. It also needs to prevent that the search depth is not bigger than the number of possible rounds that are left in the game. The adjusted epsilons function can be found in Figure 5.7 Finally an command line interface needs to be defined that is asking the human player for an move and then calling the *optimalStrategy* function to calculate the AI's move. The full implementation of this interactive version can be found in the supporting material.

Additionally to the interactive version of the connect three, an interactive version of the original connect four was also implemented. It is basically the same implementation as the connect thee version, where the rules where modified to represent the original rules

and the AI was limited to explore only six moves in advance.

```
epsilons :: [Move] -> [[Move] -> J R Move]
epsilons h = take (if (8 + length h) > (12 - length h) then 12 - length h else
    9 + length h) all
  where all = epsilon0 : epsilonX : all
        epsilonX history = epsilonMaxTupleParalell (getPossibleMoves (h ++
            history))
        epsilon0 history = epsilonMinTupleParalell (getPossibleMoves (h ++
            history))
```

Figure 5.7: Connect three interactive epsilons function

5.1.3 Discussion

Due to the simplification of the game rules, the calculation a perfect play can be done in under five seconds. The original connect four version on the other hand still needs four minutes to compute nine moves in advance. This would result in an estimated computation time of for the whole game of $2 * 10^{26}$ years computation time. A detailed discussion on how this performance was achieved and what actions where taken to improve the performance can be found in Chapter 7.2. Despite the performance issues for the whole game, the simplified connect three version shows perfectly how the selection monad enables an concise and expressive programming style. With the use of the library the implementation of the game can be done within 65 lines of code. Furthermore the interactive version shows that the library provides an easy way to either calculate a perfect play as well as with only small adjustments to the code, also a playable interactive version. It is also possible to limit the search space so that the AI is only calculating a certain number of moves in advance. When limiting the search space it is possible to implement an interactive version of the original connect four, that looks only six moves in advance instead of exploring all possible moves. The AI in this state can already be a challenging opponent for a human player. The AI's strategy is to make random moves and prevents the human player from

winning when necessary. It also takes the opportunity to make winning moves.
In the outcome function, it will only be differentiated between a win, draw or loose. This is an sufficient outcome function when the complexity of the game allows to explore all possible moves. When limiting the search space, a more sophisticated outcome function will most likely improve the AI's behaviour. When taking the current positioning at the board into the account when calculating the outcome, the AI is then always aiming to improve its current position instead of just preventing the opponent from winning as long as it has the opportunity to achieve a win on its own. However, the interactive connect four version shows that it is possible to already achieve good results with the simple win/draw/loose outcome function.

5.2 Sudoku

Sudoku is a logic based puzzle. The goal is to fill the gaps in a 9x9 matrix, so that each row and column and sub block do not contain any duplicates. The nine sub blocks are formed of 3x3 blocks in which the whole matrix is divided. The number of gaps that are to fill are defining the difficulty of the puzzle. Solving an sudoku is NP complete, however there are heuristic algorithms that can efficiently solve sudokus. The goal of this case study is to explore the monads ability to solve one player puzzles with a simple outcome function.

5.2.1 Implementation

First, the Board, Move and outcome types are defined. The board is a 9x9 matrix of integer using the third party library again. A move is a triple where the first two integers are a coordinate pointing to a position in the matrix and the third integer is the value that should be written into this field. The outcome is defining if a board is a valid solution

to the sudoku and is therefore represented as a boolean.

```
type R = Bool
type Move = (Int, Int, Int)
type Board = Matrix Int
```

Figure 5.8: Sudoku data type definition

Now a the win function needs to be defined. The win function is checking the whole board if the sudoku is still valid. Because this function is relatively time consuming to compute and it is not necessary to check the whole board if only one move is applied to a board, a second win' function is also defined, checking for a given move if this move is valid on a given board. The implementation of this win function is very expressive an will be omitted because of the limited space. In can be found in the supporting material.

```
wins :: Board -> Bool
wins' :: Move -> Board -> Bool
```

Figure 5.9: Sudku: Different type signatures of the wins functions

To define the p function, an outcome function is defined that is inserting a list of moves in a given board and checking for each insertion if the board is still valid. An starting board is also defined, where some numbers are already inserted into an empty board to form the initial puzzle. The implementation of the p and *outcome* functions look like this:

```
p :: [Move] -> R
p ms = outcome ms startingBoard

outcome :: [Move] -> Board -> R
outcome [] b    = wins b
outcome (x:xs) b = let nb = insert x b in if wins' x b then outcome xs nb else
    wins nb
```

Figure 5.10: Sudoku outcome function

To describe the AI that solves the sudoku, the maximum function on boolean is used. The AI wants to make only moves that produces an valid sudoku in the end. Therefore an infinite list of selection functions is created and then reduced to a list of same length as there are gaps in the sudoku. The *startingMoves* is a list of moves, that contain the initially defined values that are necessary to define the puzzle. The implementation of the *epsilons* function looks like this:

```
epsilons :: [[Move] -> J R Move]
epsilons = take 12 all
  where all = epsilon' : all
        epsilon' history = epsilonMaxBool (getPossibleMoves (startingMoves ++
           history))
```

Figure 5.11: Sudoku epsilons function

5.2.2 Discussion

This case study shows that it is generally possible to solve puzzles like sudoku with the help of the selection monad. However, the algorithm that is implemented by the library to solve this game (Minimax) is growing expectationally with each layer of moves. Therefore the case study implementation is able to solve sudokus with 12 gaps in about 100 seconds, but every additional gap increases the computation time by the factor of approximately 3. A detailed discussion of the performance of this implementation can be found in Chapter 7.2. Algorithms that are specialised to solve sudokus, for example using an backtracking algorithm or stochastic approaches [8, 9], perform significantly better than the case study implementation with the selection monad. It therefore shows that is generally possible to solve puzzle like sudoku with the selection monad, however this approach is not suitable to solve sudokus with more than 15 gaps. Similar to the connect three implementation as described in Section 5.1.3, the use of the selection monad in combination with the library enable a concise implementation in around 100 lines of code. Only the board and a function that checks the validity of the board needs to be defined.

5.3 Simplified Chess

Chess is played by millions of players worldwide. Many computer scientist are working since the 1950ties on chess AI's in order to build the best chess AI. Many of this chess AI's are based on the minimax algorithm and the selection monad should therefore be suitable to implement an chess AI. However, to limit the complexity of the game for this case study, only a simplified chess version was implemented, focusing on solving end-games and only supporting the following chess pieces: King, Queen, Rook and bishop.

5.3.1 Implementation

As before, the board, move and outcome types need to be defined. The chessboard is represented using a matrix of chess pieces, where a chess piece is either nothing or a Queen, Rook, King or Bishop of a specific colour. The colour is either white or black. A move is a tuple of two positions on the board. The first element is the origin position and the second is the target position. A position is a tuple of integer representing the x and y values. The outcome type is like in the connect four implementation a tuple of integer where the first integer is tracking if the game is won, lost or a draw, and the second integer is counting the moves that need to be done to achieve this outcome. The definition of this data types is de following:

```
type R = (Int, Int)
type Position = (Int, Int)
type Move = (Position, Position)
type Board = Matrix ChessPice
data Colour = W | B
  deriving (Eq, Show)
data ChessPice = N | Queen Colour | King Colour | Rook Colour | Bishop Colour
  deriving (Eq, Show)
```

Figure 5.12: Chess data type definitions

As this simplified version of chess is not supporting a draw, the win function can be easily

implemented by checking the whole board if there is still a king on the board for the given colour. The implementation of this win function looks like this:

```
wins :: Board -> Colour -> Bool
wins b c = King (changeColour c) `notElem` toList b
```

Figure 5.13: Chess wins function

In the contrast to the simple **wins** function, the function calculating all possible moves is basically implementing all the chess rules. In order to do so it is iterating through the whole board and combining all possible moves of each peace with the colour of the current player. The implementation of this function can be seen in Figure 5.14. The implementations that calculate all possible moves for each chess piece are really expressive and will be omitted because of the limited space.However, the full implementation can be found in the supporting material.

```
getPossibleMoves :: [Move] -> [Move]
getPossibleMoves m = concat [getMoves (x,y) b c | x <- [1..8], y <- [1..8],
    getMoves (x,y) b c /= [] ]
  where
    b = insertMoves startBoard m
    c = if (length m `mod` 2) == 1 then B else W

getMoves :: Position -> Board -> Colour -> [Move]
getMoves p b c1 = case b ! p of
                    N         -> []
                    (King c2)  -> if c1 == c2 then getKingMoves p else []
                    (Queen c2) -> if c1 == c2 then getQueenMoves p b else []
                    (Rook c2)  -> if c1 == c2 then getRookMoves p b else []
                    (Bishop c2) -> if c1 == c2 then getBishopMoves p b else []
```

Figure 5.14: Functions that calculate all possible moves of the current player in chess.

To calculate the outcome of the chess game, a list of moves will be applied to a previously defined start board. Therefore a outcome helper function is defined, calculating the outcome based on the current colour, a list of moves, the board this moves need to be

inserted to and a count of previous moves.

```
value :: Board -> Int
value b | wins b W = 1
        | wins b B = -1
        | otherwise = 0

outcome :: Colour -> [Move] -> Board -> Int -> R
outcome _ [] b i     = (value b, i)
outcome c (m : ms) b i = let nb = insert b m in
                         if wins nb c then (value nb, i+1) else outcome
                             (changeColour c) ms nb (i+1)

p :: [Move] -> R
p ms = outcome W ms startBoard 0
```

Figure 5.15: Chess outcome function

The *eplsions* function is the same as in the connect three version, describing the minimax algorithm with alternating minimum and maximum function. It is limited to search only five moves in advance. This parameter needs to be adjusted to fit each endgame individually.

```
epsilons :: [[Move] -> J R Move]
epsilons = take 5 all
  where all = epsilon0 : epsilonX : all
        epsilonX history = epsilonMinTupleParalell (getPossibleMoves history)
        epsilon0 history = epsilonMaxTupleParalell (getPossibleMoves history)
```

Figure 5.16: Chess epsilons function

5.3.2 Discussion

The implementation is able to solve a simple endgame with a checkmate in six moves in under a minute. Every additional move that is computed in advance increases the computation time by the factor of 14. A detailed discussion about the performance of this implementation can be found in Chapter 7.2. The simplified version of chess shows that it

is possible to implement chess with the help of the selection monad. The implementation of the simplified rules as well as the AI is with around 140 lines of code really concise. As the outcome function is only considering a wins, draws and looses the implantation can only solve end games that are lead to a win in six moves in a reasonable amount of time. In general, this case study shows that even sophisticated games can be implemented to some extend with the selection monad. It also enables an elegant way to define the behaviour of the AI. In order to expand this case study to a chess AI that is challenging to play against, all chess rules need to be implemented and the outcome function should be expanded to a function that takes the current positioning on the board into account.
When comparing the approach to other implementations of Chess AI's, many implementations also use the minimax algorithm. To reduce the search space the minimax algorithm is often combined with alpha/beta pruning, which could be a possible extension to the library and is discussed further in Chapter 7. The key element for making a good Chess AI is the outcome function. Having an efficient outcome function that is also taking the current positioning into account increases the difficulty of the AI significantly. This kind of AI's can be easily done with the selection monad.

Chapter 6

Performance and Testing

In order to measure the performance of the case studies, each case study was tested with an example input and the computation time was measured for layer of moves they compute in advance. The computation time grows exponentially with each layer of moves that is added to the computation. Therefore there will always be a limit from which on it not feasible any more to add another layer of moves. The only chance to achieve good results is therefore to push this limit as far back as possible. As a result of the performance analysis three major factors that influence the performance where identified. First the amount of moves that are possible at each layer, second the complexity of the functions that calculate the outcome and the possible moves and third an efficient minimum and maximum computation using parallelism.

The following Chapter will go through each case study and explains the actions that where taken to increase the performance of the implementation in respect to the previously introduced factors that influence the performance. For each step of the optimisation the performance was measured and is displayed in a graph.

All performance measurements where executed on the same machine with an AMD FX-8350 Eight core Processor with 4.00 GHz and 16 GB of RAM. The different versions with different optimisations can be found in the supporting material.

The final section then explains how the library was tested.

6.1 Connect four

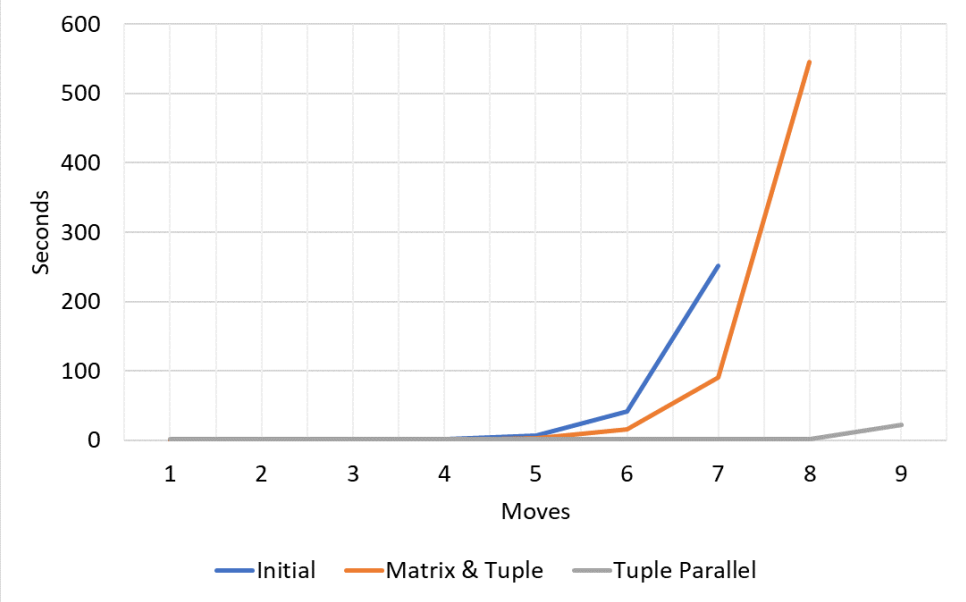

Figure 6.1: Performance of the different Connect Three implementations

A quick performance analysis of an initial implementation of connect four showed that due to the exponentialy growing computation type it is not feasible to explore all possible moves. To reduce the search space the simplified Connect Three version of the game was introduced. While an initial verison of connect four was able to consider 5 moves in advance under 100 seconds as seen in Figure 6.2, the simplified connect thee version was already able to consider 6 moves in advance in under 100 seconds as seen in Figure 6.1. The following performance optimisations where applied to both versions.

6.1.1 Matrix instead of Lists

In order to improve the performance further, a matrix library was used instead of nested lists to represent the game board. This library enables access times of $O(1)$. This in

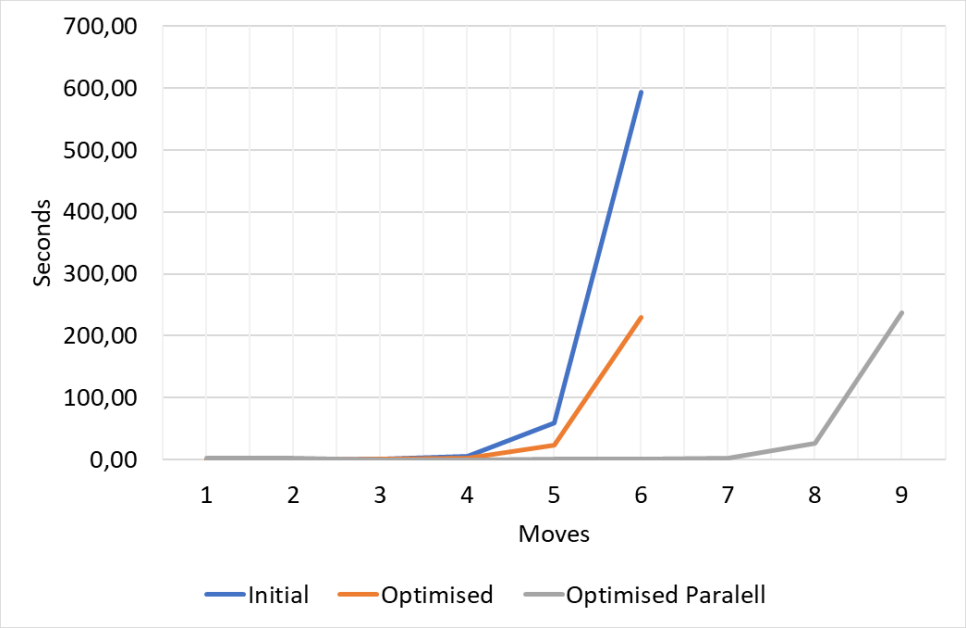

Figure 6.2: Performance of the different Connect Four implementations

combination with small adjustments to the function that calculates the outcome of the game increased the performance slightly. The increased performance can be seen in Figure 6.2 and Figure 6.1.

6.1.2 Using parallel min max

To increase the performance further a parallel version of the minimum and maximum function was used which leads to a huge performance increase. The parallel version of connect four was able to consider 8 moves in advance in under 100 seconds instead of 5 moves and in the connect three version it where even 9 moves instead of 6.

6.1.3 Discussion

With this performance optimisations it was possible to compute an perfect play for the simplified connect three version in 22 seconds. This shows that an optimised outcome function and a reduction of the possible move for each layer improves the performance

slightly and using parallelism to compute the minima and maxima has a huge impact to the performance. Unfortunately even for the optimised parallel connect four version the estimated time of calculating an perfect play is with $2 * 10^{26}$ years still not feasible. However, the interactive version of connect four showed that it is possible to achieve good results when only considering 6 moves in advance. Therefore the selection monad is suitable to build an interactive version of Connect Four and is even able to calculate a perfect play for the simplified connect three version.

6.2 Sudoku

Solving sudokus is known to be NP complete. However there exist some heuristc and stochastic approaches [9, 8] that can solve sudokus relatively efficient. Compared to this algorithm, the case study implementation is doing a brute force approach that tries all possible combinations. This approaches results in an exponential computation time. As shown in Figure 6.3 the unoptimised version can solve a sudoku with 11 gaps in under 100 seconds where the optimised version can solve sudokus with 12 gaps in under 100 seconds.

6.2.1 Optimised version

In order to optimise the computation, the function that calculates if a sudoku is valid was optimised. Instead of checking the validity of the whole board after inserting a move into the board, only the row, colum and square in which a number was inserted is checked for validity.

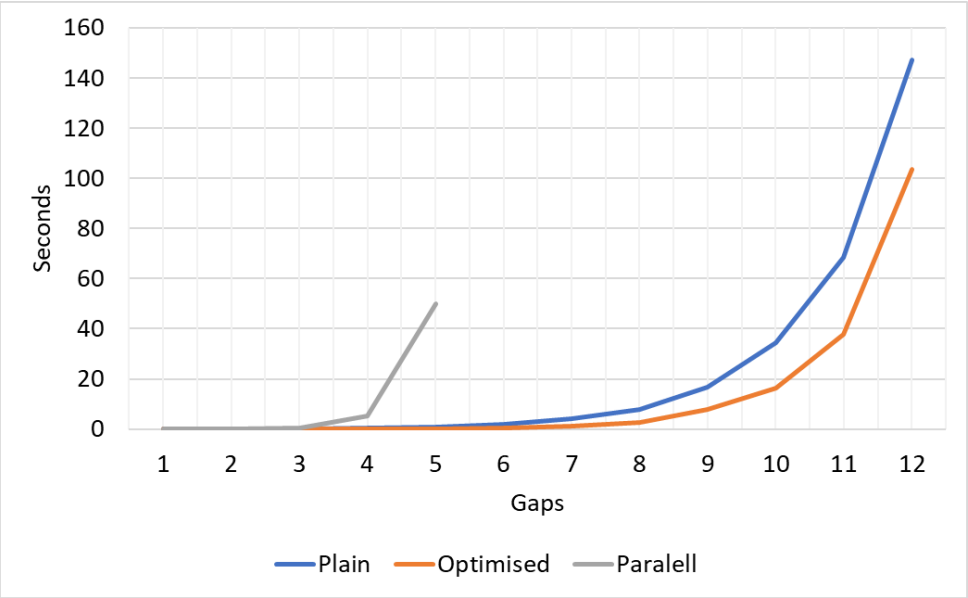

Figure 6.3: Performance of the different Sudoku implementations

6.2.2 Parallel version

In the case of sudoku, the parallel version of the implantation did significantly worse than the other implementations. A reason for this might be that the minimum and maximum functions that work for boolean as outcome type are only searching the list of moves until they find the correct move. In this case the pruning inside the boolean function is more efficient than using an parallel version of the maximum function. Also, the parallel version is only using around 3% to 5% of the CPU where in the other case studies the CPU was used up to 100%.

6.2.3 Discussion

The sudoku case study strengthen the Hypothesis that an efficient outcome function has an impact on the performance. However the simple brute force approach that the selection monad provides is not good enough to solve sophisticated sudoku puzzles. It also shows that the right choice of minimum and maximum function can have a huge impact on the performance (See Figure 6.3). The functions that is optimised for Boolean work

significantly better than the generic parallel version. As the selection monad approach only enables this brute force approach and does not allow to customise the AI's strategy to something that works well for sudokus, the selection monad might not be suitable to solve sophisticated sudoku puzzles.
However this case study shows that it is generally possible to solve single player puzzles with an brute force approach and the only limitation is the complexity of this puzzles. To resolve this performance issue, the selection monad could be modified that it supports some sort of pruning (like alpha-beta pruning). However as this pruning optimisation is not in the scope of this thesis, it will briefly discussed in the Future Work Section 7.4.

6.3 Chess

Building Chess AI's was always a challenge in computer science. The complexity of the game is huge, and experts are not sure about it can be solved (If a perfect play can be found). Because of the general complexity of the game only a simplified version of chess was implemented in the case study. This simplified version has an easy outcome function that is basically only checking if both kings are still on the board. However, the huge amount of possible moves at each stage of the game affecting the performance of this implementation the most. As the different performance optimisations that can be done to increase the performance are already studied in detail in the other two case study, only one implementation of the chess game exist. The aim of this case study is less to explore the performance problems of selection monad implementations and more to show that even sophisticated games like chess can be implemented to some extend with the help of the selection monad.

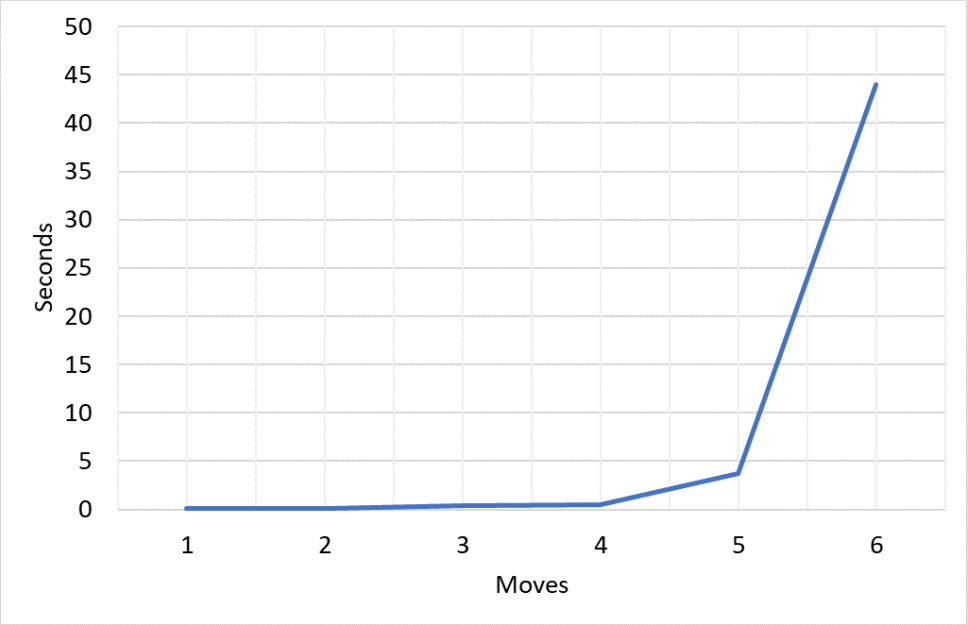

Figure 6.4: Performance of the simplified Chess implementations

6.3.1 Runtime

The performance of the Chess implementation, as seen in Figure 6.4, suggested that with the help of the the chess AI can only calculates up to six moves in advance. As the outcome function is already as simple as possible, this performance is due to the huge amount of moves that add to the computation with each layer.

6.4 Discussion

In order to improve the performance, the selection monad needs to be extended to support some sort of pruning (like Alpha-Beta pruning). As this pruning optimisation is not in the scope of this thesis, it will briefly discussed in the Future Work Section 7.4. However, this implementation is already able to solve simple endgames, considering 6 moves in advance in under 100 seconds. Considering 7 moves in advance already increases the computation time to over 1000 seconds. With a more sophisticated outcome function, this AI can easily extended to an interactive AI that calculates up to 6 moves in advance. This shows that

the selection monad is already able to support an implementation of a chess AI, and with some pruning extensions to the selection monad the number of moves that are calculated in advance might increase further.

6.5 Testing of the library

The selection monad library is tested with unit tests using the HUnit test library [10]. This Unit tests are covering the different minimum and maximum functions, as well as some helper functions for the selection monad. The functions that calculate the optimal play need an actual game implementation and are therefore covered and tested with the case studies. An explaination how to execute the tests, as well as the source code of the tests can be found in the supporting materials.

Chapter 7

Discussion and Outlook

This Chapter briefly summarises what was done during the masters project in Section 7.1 and discusses the major performance issue that occurred during the implementation of the case studies in Section 7.2. Section 7.3 is summarising the results of the case studies and finally Section 7.4 is outlining what could be done as future work.

7.1 Summary

At the beginning of the project, a literature research was done regarding the selection monad. During this phase, a deep understanding of the mechanics of the selection monad was gained. With this knowledge, the three case study games where implemented. In order to increase the performance of the case studies, performance optimisation where applied to the outcome functions of the case studies. Simultaneously, the library was built, summarising common functionalities of the case studies. Also, some performance optimisations to the minimum and maximum functions where implemented in the library. During this implementation phase, a general knowledge in Game Theory was gained, especially regarding to implementing game AI's in general. Furthermore, some general knowledge in library design was gained during the library building process.

Finally, a performance benchmark on the case studies with their different optimisations was done in order to identify the major factors that influence the performance of the implementation the most.

7.2 Performance

The complexity of the games was identified as a major factor when deciding whether the selection monad is a suitable approach to implement an AI for this game or not. Due to the exponential computation time of the minimax algorithm (the underlying algorithm when using the selection monad), currently only way to increase the performance is to provide a fast implementation of the outcome function and the use of the right minimum and maximum function which ideally parallelise the computation.

7.3 Conclusion

The selection monad provides a elegant way of implementing sequential games and puzzles. The case studies show that the implementation of an sequential game can be done in under 150 lines of code. Furthermore, with library a programmer does not need to care about how to implement the AI. The AI can be implemented by just defining an outcome function and using the provided minimum and maximum functions. While implementing sequential games the complexity of the games resulting in performance problems was identified as the only issue. This issue may be partially resolvable by building some sort of pruning (like alpha-beta pruning) into the library.

However, summarising common functionalities into a library improves the modularity of case studies code significantly, and future programmers profit from the performance optimisations that are build in the minimum and maximum functions. They therefore only need to choose the right functions that fit their outcome type.

7.4 Future work

To extend the functionality of the library, a possible future work could be implementing more case studies to explore a more diverse set of games. In this thesis only games with perfect knowledge (Games where every player is perfectly informed about all events and moves that where done previously) where considered. Therefore a possible future work could explore games without perfect knowledge where some part of the game is hidden to the player. This for example could be a card game or mastermind. Furthermore, a possible future case study could explore the ability of the selection monad to play games that are based on some sort of chance. An example for this kind of games could be Black Jack.

Another future work could explore some performance optimisations that could be applied to the library. For example some sort of pruning could limit the search space of moves that need to be explored. Alpha-Beta pruning is an pruning algorithm that is skipping whole branches of the tree during the minimax algorithm. It skips the evaluation of a branch in the tree when at least one possible move has been found that show that this branch is worse than the previous explored moves.

Bibliography

[1] Escardó Martin; Oliva Paulo. What sequential games, the Tychonoff theorem and the double-negation shift have in common . *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming. ACM, 2010. S. 21-32.*, 2010.

[2] Martin Escardó; Paulo Oliva. Compainion Haskell programms to the: "What sequential games, the Tychonoff theorem and the double-negation shift have in common" paper. Online; `https://www.cs.bham.ac.uk/~mhe/papers/msfp2010/MSFP2010/haskell/modular/monadic/`, July 2010.

[3] John Tromp. Number of legal 7 x 6 connect-four positions. Online; `https://oeis.org/A212693`, May 2012.

[4] Bryan O'Sullivan; John Goerzen; Donald Bruce Stewart. *Real world haskell: Code you can believe in.* O'Reilly Media Inc., 2008.

[5] Victor Allis. *A Knowledge-based Approach of Connect-Four.* PhD thesis, Vrije Universiteit Amsterdam, The Netherlands, 1988.

[6] John Tromp. John's Connect Four Playground. Online; `https://tromp.github.io/c4/c4.html`.

[7] Daniel Díaz. A native implementation of matrix operations. Online; `https://hackage.haskell.org/package/matrix-0.3.6.1`, March 2018.

[8] Rhyd Lewis. Metaheuristics can Solve Sudoku Puzzles. . *Journal of Heuristics, vol. 13 (4), pp 387-401.*, 2007.

[9] Daniel Díaz. Brute Force Search. Online; `http://intelligence.worldofcomputing.net/ai-search/brute-force-search.html#.W3Mng-hKguV`, December 2009.

[10] Dean Herington. HUnit: A unit testing framework for Haskell. Online; `http://hackage.haskell.org/package/HUnit-1.6.0.0`, March 2017.