Check This!
A Literate Haskell Program to Play Checkers

Thomas Parslow
twp21@sussex.ac.uk

February 28, 2008

Contents

Chapter 1

Introduction

This document describes–and in fact is–a program to play checkers written in the Haskell[1] programming language. This document is a *Literate Haskell* program, this means that its source code is both a valid a LaTeX document and a valid Haskell program.

gtk2hs[2] is used for the GUI so it should have no problems running on Windows, OSX or any flavour of Unix. However it has only been tested on Linux (Ubuntu 7.10 with GHC 6.6.1).

The modules that make up the program are:

Board (§2.1, p 4) Defines a data structure to represent a checkers board. It also defines data structures to represent locations on the board and "location deltas".

Hops (§2.2, p 9) Defines "hops" which are the moves which make up a turn.

GameState (§2.3, p 12) Defines a game state data structure, a successors function, and, a function to check if either side has won.

Negamax (§3.1, p 14) Defines a generic version of the Negamax algorithm, not specific to the game of checkers.

EvalFuns (§3.2, p 15) Defines static evaluation functions and functions to combine them. These can be used as heuristic functions for the Negamax algorithm.

Strategies (§3.3, p 16) Uses the Negamax algorithm and the evaluation functions defined in `EvalFuns` to create an AI strategy.

ANNEvalFun (§4.1, p 17) An alternative method of creating evaluation functions using a multi-layer feed-forward Artificial Neural Network (ANN).

EvolveANN (§4.2, p 18) A Genetic Algorithm to evolve a set of weights for the ANN defined in `ANNEvalFun`.

GUI (§5.1, p 21) A GUI based on the GTK tool kit. This acts as the main module of the program.

[1] Haskell is a lazy functional language: `http://www.haskell.org/`
[2] `http://www.haskell.org/gtk2hs/`

The program can be built using the GHC[3] Haskell Compiler but may well work with Hugs[4] as well. The command to build the program for normal GUI operation is:

```
ghc -O2 --make GUI.lhs
```

Before the PDF version of the program source code can be generated all *.lhs* files must be sym-linked so that they can also be accessed using the *.tex* extension. Once this is done the following commands can be run.

```
pdflatex checkers.tex
makeindex checkers
pdflatex checkers.tex
```

[3]The Glasgow Haskell Compiler: `http://www.haskell.org/ghc/`
[4]http://www.haskell.org/hugs/

Chapter 2

Game and Board Representation

2.1 The Board Data Structure

The first thing that's needed is a data structure to represent a Checkers board. This module will confine itself to representing the board and locations on the board, it will not represent movement.

Using the exported functions it is possible to:

- Create a `Board` containing the initial state of a checkers board
- Enumerate all valid locations (represented by a `Location` data type)
- Get and set a given location's contents
- Transform a location into another by way of a `LocationDelta` representing one of the 4 possible (diagonal) directions.
- Get the row and column coordinates of a `Location`
- Find out if a given location is on a king row

Conspicuously absent from this list is a way to construct a `Location` from row and column coordinates, this is intentional and helps insure that an inconsistent board state can never be constructed. Any function that could return an invalid `Board` or `Location` will have a `Maybe` return type and will return a `Nothing` in place of invalid data.

2.1.1 Interface

The module exports the following data types and functions.

```
module Board (
             Board,
```

Data type representing a checkers board (§2.1.2, p 6)

```
             Location,
```

Data type representing a location on a checkers board (§2.1.2, p 6)

```
Side(Black, White),
```

Enumeration type indicating the side of checkers player (§2.1.2, p 6)

```
CheckerType(King, Normal),
```

Enumeration type indicating the type of a checkers piece (§2.1.2, p 6)

```
Checker(Checker),  -- :: Side → CheckerType → Checker
```

A data type consisting of a `Side` and `CheckerType` used to represent a checker piece (§2.1.2, p 6)

```
SquareContent,
```

An alias for `Maybe Checker` used to represent the contents of a given square on a `Board` (§2.1.2, p 6)

```
row,               -- :: Location → Int
```

Retrieve the row coordinate from a `Location` (§2.1.2, p 7)

```
col,               -- :: Location → Int
```

Retrieve the col coordinate from a `Location` (§2.1.2, p 7)

```
startBoard,        -- :: Board
```

A constant set to the starting board state for checkers (§2.1.2, p 7)

```
getSquare,         -- :: Location → Board → SquareContent
```

Gets the contents of the square on a `Board` at a given `Location` (§2.1.2, p 7)

```
setSquare,         -- :: Location → SquareContent → Board → Board
```

Returns a copy of a `Board` with contents of the square at a given `Location` set to the new value supplied (§2.1.2, p 7)

```
allLocations,      -- :: [Location]
```

A list of all valid `Location`s. The list covers all possible locations on a checkers board (all the black squares) (§2.1.2, p 8)

```
allSquares,        -- :: Board → [(Location, SquareContent)]
```

A convenience function that returns all possible `Location`s plus the contents of those `Location`s for a given board (§2.1.2, p 8)

```
allSquaresForSide, -- :: Board → Side → [(Location, SquareContent)]
```

As for `allSquares` except only returns squares occupied by a given side (§2.1.2, p 8)

```
validDeltas,       -- :: Checker → [LocationDelta]
```

Returns all the valid directions a given checker piece can move in—2 for a normal checker piece and 4 for a king (§2.1.2, p 8)

```
applyDelta,        -- :: LocationDelta → Location → Maybe Location
```

Apply a `LocationDelta` to a `Location` producing a `Just Location` if the result of adding would be a legal position and `Nothing` if it would not. (§2.1.2, p 8)

```
              kingLocation,      -- :: Location → Bool
```

A predicate indicating whether a given `Location` is on a king row or not (§2.1.2, p 8)

```
              opposition,        -- :: Side → Side
```

Converts `Black` to `White` and `White` to `Black` (§2.1.2, p 9)

```
              swapSides          -- :: Board → Board
```

Swaps the sides on a `Board` and rotate it 180° (§2.1.2, p 8)

```
             ) where
import Data.Maybe (catMaybes, maybeToList)
```

2.1.2 Code

Data Types

A checker can be either black or white and is either a king or a non-king (which I will call "normal") so I'll start by defining enumerations to represent those two properties.

```
data Side = White | Black deriving (Eq, Show)

data CheckerType = King | Normal deriving (Eq, Show)
```

I can then define the checker type in terms of those two enumerations.

```
data Checker = Checker {side        :: Side,
                        checkertype :: CheckerType} deriving (Eq,Show)
```

A given square can be either empty, or, it can contain a checker. The obvious way to do this is to use a Maybe type so that's what I do. A type synonym is defined to make the type signatures clearer.

```
type SquareContent = Maybe Checker
```

A type to represent the board itself can now be defined. If the white squares are ignored this can simply be a 32 element list. The implementation is hidden from other modules so it could be swapped out for a more efficient one later on.

```
data Board = MkBoard [SquareContent]  deriving (Eq, Show)
```

A special data type is created to represent locations on the board, this means that we can insure that an invalid location can never be constructed. It also means that an efficient internal representation could be used later on if it needed.

Locations are specified using full board coordinates (with the white squares.)

```
data Location = MkLoc Int Int deriving (Eq, Show)
```

A representation of movement will also be useful.

```
data LocationDelta = MkLocDelta Int Int deriving (Eq, Show)
```

Constructors

A function is also required to create a starting board. I have chosen to provide a function, `startBoard`, that creates a board set up for the beginning of a game of checkers, I choose this over a blank board as I think it will be more convenient—it would be very rare to want a blank starting board.

```
startBoard :: Board
startBoard = MkBoard $ (replicate 12 $ Just $ Checker Black Normal) ++
                       (replicate 8 $ Nothing) ++
                       (replicate 12 $ Just $ Checker White Normal)
```

A `makeLocation` function is provided to construct a location given row and column coordinates. It returns a `Just location` if the coordinates are valid and `Nothing` if not.

This function was originally part of the public interface to the module but no longer is. I found that use of the `LocationDeltas`, allowed manipulation of locations in a safer way.

```
makeLocation :: Int → Int → Maybe Location
makeLocation r c
    | r ≥ 0 && r < 8 && c ≥ 0 && c < 8 && (r ‘mod‘ 2) ≠ (c ‘mod‘ 2) = Just $ MkLoc r c
makeLocation _ _ = Nothing
```

Accessors

Using these location data types we can get and set locations on the board. A internal helper function, `getIdx`, is used to translate a `Location` into an index into the list holding the board state. Since only valid locations can ever be constructed we don't need to worry about the row or column being out of bounds.

```
-- internal module function to get index for position
getIdx :: Location → Int
getIdx (MkLoc row col) = row*4+(col ‘div‘ 2)

getSquare :: Location → Board → SquareContent
getSquare loc (MkBoard boardLst) =
    let i = getIdx loc in
    boardLst !! i

setSquare :: Location → SquareContent → Board → Board
setSquare loc content (MkBoard boardLst) =
    let i = getIdx loc in
    MkBoard $ (take i boardLst) ++
              [content] ++
              (drop (i+1) boardLst)

row, col :: Location → Int
row (MkLoc r _) = r
col (MkLoc _ c) = c
```

Location transformations

Once we have a location we will want to transform it into other locations. This will be done using `LocationDelta` data. This means that most parts of the program need not concern themselves with actual board coordinates.

First we need a way to construct `LocationDelta`s, this is fairly easy as only a few will ever be needed—just those representing valid moves on the checker board. This will be done using a function that takes a `Checker` and returns `LocationDelta`s representing all the moves it could make.

```
validDeltas :: Checker → [LocationDelta]
validDeltas (Checker _ King) = validDeltas (Checker White Normal) ++
                               validDeltas (Checker Black Normal)
validDeltas (Checker side Normal) = [MkLocDelta (dir side) (-1),
                                     MkLocDelta (dir side) 1]
    where dir Black = 1
          dir White = -1
```

A function is then required to apply a delta to an existing location, it's return type is `Maybe Location` so that `Nothing` can be returned if applying the delta would result in invalid coordinates.

```
applyDelta :: LocationDelta → Location → Maybe Location
applyDelta (MkLocDelta rd cd) (MkLoc r c) = makeLocation (r + rd) (c + cd)
```

Board transformations

A function to swap the sides on the board will be very useful for some board evaluation functions. This means to turn it 180° and switch the colours. This means that a board evaluation function which assumes it is evaluating for black can be used for white as well.

```
swapSides :: Board → Board
swapSides board = foldr (uncurry setSquare) board squares
    where squares = [(MkLoc (7-r) (7-c),Just (Checker (opposition s) t))|
                      (MkLoc r c,Just (Checker s t)) ← allSquares board]
```

Board information

A constant containing all valid locations on a checkers board will be useful, this can be used in conjunction with `getSquare` to get all pieces on a board.

```
allLocations :: [Location]
allLocations = catMaybes [makeLocation r c| r  ← [0..7], c ← [0..7]]
```

A couple of helper functions to return all squares on a board along with their location and all squares containing pieces of a given side will simplify other code later on.

```
allSquares :: Board → [(Location, SquareContent)]
allSquares board = [(loc, getSquare loc board) | loc ← allLocations]

allSquaresForSide :: Board → Side → [(Location, SquareContent)]
allSquaresForSide board sid =
    filter ((==[sid]) ∘ map side ∘ maybeToList ∘ snd) (allSquares board)
```

A predicate function to check if a location is a "king" location will also be useful. A "king" location is a square which moving into will cause a normal checker to become a king, that is to say it is all the locations on the top and bottom lines of the board.

```
kingLocation :: Location → Bool
kingLocation (MkLoc 0 _) = True
kingLocation (MkLoc 7 _) = True
kingLocation _ = False
```

We will also often want a function to flip a side, given `White` it would return `Black` and vise versa.

```
opposition :: Side → Side
opposition White = Black
opposition Black = White
```

2.2 The Hop Data Structure

Building on the `Board` module (§2.1, p 4) we next want to represent what I call “hops”. A hop is a a single move of a checker piece from one board location to another—possibly capturing another piece along the way. A hop is not always a complete move, several capture hops may be strung together to form one move.

2.2.1 Interface

```
module Hops(
            Hop(SingleHop,
                CaptureHop,
                source,
                capture,
                destination,
                newBoard),
```

Data type representing a hop, the constructors—`SingleHop` and `CaptureHop`–are a public part of the module interface to allow easy pattern matching (§2.2.2, p 9)

```
            allowedHops,   -- :: Board → Side → [Hop]
```

Returns all legal `Hops` for a given `Side` on a supplied `Board` (§2.2.2, p 10)

```
            continuingHops -- :: Hop → [Hop]
```

Returns all legal hops continuing hops from a given `Hop`. Always returns the empty list for a non-capture hop (§2.2.2, p 10)

```
           ) where
import Board
import Data.Maybe (maybeToList, isNothing)
import Control.Monad (guard)
```

2.2.2 Code

Data Types

A non-capturing hop is represented by a source location, a destination location, and, a new board state. A capturing hop is the same as a non-capturing hop except that it has a field for the captured piece as well.

```
data Hop = SingleHop {kingHop      :: Bool,
                      source       :: Location,
                      destination  :: Location,
                      newBoard     :: Board}
         |
           CaptureHop {kingHop     :: Bool,
```

```
                 source      :: Location,
                 capture     :: Location,
                 destination :: Location,
                 newBoard    :: Board} deriving (Eq, Show)
```

Interface functions

I'm going to start by defining the two functions which form the interface to this module, followed by the helper functions they in turn rely on.

The first of these two is the `allowedHops` function, its job is to return a list of valid hops given a current board state and the side whose turn it is. It is very simply defined in terms of two functions—`captureHops` (§2.2.2, p 11) and `singleHops` (§2.2.2, p 11)—which will be defined later on. It neatly expresses the rule that the legal hops are any capture hops unless there are no legal capture hops, in which case the legal hops are all legal single hops—that is to say that capture hops are forced.

```
allowedHops :: Board → Side → [Hop]
allowedHops board side = case captureHops board side of
                           [] → singleHops board side
                           x  → x
```

As a hop is only a partial turn a function is needed to find the allowed continuations of a hop. It makes use of `captureHopsForLocation` (§2.2.2, p 11) which is similar to `captureHops`—in fact the latter is implemented in terms of the former as we will see below—except that it only returns capture hops starting from a given square.

The rule that if a checker becomes a king it must stop its turn immediately is also representing here (by pattern matching `False` on the `kingHop` field of the previous `Hop`).

```
continuingHops :: Hop → [Hop]
continuingHops (CaptureHop False _ _ dest board) = captureHopsForLocation board dest
continuingHops _ = []
```

A Quick Detour: The List Monad and "do" Notation

The functions below make use of the `List Monad`[1] to find all possible hops. "do" notation is used to keep the code pretty. Monads are really not as scary as they sound and the `List Monad` is certainly the least scary of the bunch. A quick example of the `List Monad` and "do" notation might help:

```
example :: [Int] → [Int] → [Int]
example as bs = do a ← as
                   b ← bs
                   guard $ a ≠ b
                   return $ a+b
```

Is equivalent to the following (untested) Java code (ignoring the issues of lazy vs. strict evaluation):

```
int[] example(int[] as, int[] bs) {
    ArrayList<int> = new ArrayList<int>();
    for(int i = 0; i < as.length; i++) {
```

[1] See `http://www.haskell.org/all_about_monads/html/listmonad.html` for more information. See also `http://lukeplant.me.uk/blog.php?id=1107301643` for an easier introduction if you're already familiar with Python's List Comprehensions.

```
        for(int j = 0; j < bs.length; j++) {
          if (!(as[i] != bs[j])) {
              continue;
          }
          retvals.add(as[i] + bs[j]);
        }
    }
    return (int[]) retvals.toArray(new int[retvals.size()]);
}
```

As you can see, this makes for a powerful abstraction in certain situations.

Helper Functions

The first helper function is `singleHops`, this function returns all legal single hops for a given `Board` and `Side`. It does this by first taking all squares containing pieces of the correct colour. Next it applies all valid deltas for the given piece to its location—deltas that would move the piece of the board are ignored since `applyDelta` would return `Nothing` which `maybeToList` converts to the empty list. All destinations that would move a piece onto another piece are discarded by the guard and the remaining ones used to construct `Hop`s using the `SingleHop` constructor.

```
singleHops :: Board → Side → [Hop]
singleHops board side = do
  (source, Just checker@(Checker _ checkertype)) ← allSquaresForSide board side
  delta ← validDeltas checker
  dest  ← maybeToList $ applyDelta delta source
  -- Make sure the hop is valid for the board
  guard $ isNothing $ getSquare dest board
  let newBoard = setSquare source Nothing ∘
                 setSquare dest   (Just $ maybeBecomeKing checker dest) $ board
  return $ SingleHop (kingLocation dest && checkertype ≠ King) source dest newBoard
```

The `captureHops` function returns all valid capturing hops, it does this by calling the `captureHopsForLocation` function for each piece of the correct colour on the board.

```
captureHops :: Board → Side → [Hop]
captureHops board side = do
  (source, _) ← allSquaresForSide board side
  captureHopsForLocation board source
```

`captureHopsForLocation` works in a similar way to `singleHops`, using the `List Monad` to generate then filter all possible moves—except this time it's only considering a single starting `Location`.

```
captureHopsForLocation :: Board → Location → [Hop]
captureHopsForLocation board source = do
  let Just checker@(Checker side checkertype) = getSquare source board
  delta ← validDeltas checker
  cap   ← maybeToList $ applyDelta delta source
  dest  ← maybeToList $ applyDelta delta cap
  -- Make sure the hop is valid for the board
  guard $ isNothing $ getSquare dest board
  Checker capside _  ← maybeToList $ getSquare cap board
```

```
  guard $ capside ≠ side
  let newBoard = setSquare source Nothing ∘
                 setSquare cap    Nothing ∘
                 setSquare dest   (Just $ maybeBecomeKing checker dest) $ board
  return $ CaptureHop (kingLocation dest && checkertype ≠ King) source cap dest newBoard
```

The `maybeBecomeKing` function was used above to get the checker piece to place in the new location. This will usually be the same as the checker from the source location except in the case of a normal checker moving into a king square, in this case the checker will become a king.

```
maybeBecomeKing :: Checker → Location → Checker
maybeBecomeKing (Checker side _) dst | kingLocation dst = Checker side King
maybeBecomeKing checker _ = checker
```

2.3 The GameState Data Structure

Building on the data types and associated helper functions in the `Board` and `Hops` this module defines a data structure to encode the current game state and a function to generate successor states. It also defines the starting state as a constant—`startState` (§2.3.2, p 13).

2.3.1 Interface

```
module GameState(GameState(GameState,
                           prevState,
                           hops,
                           board,
                           turnNext),
```

A data structure to hold current game state (§2.3.2, p 13)

```
                 startState, -- :: GameState
```

A constant holding the starting game state (§2.3.2, p 13)

```
                 successors, -- :: GameState → [GameState]
```

The successor function for `GameState`s, returns a list of possible successor states (§2.3.2, p 13)

```
                 winner      -- :: GameState → Maybe Side
```

A predicate which indicates whether a side has won in the given `GameState`, returns `Nothing` if neither side has won (§2.3.2, p 13)

```
                ) where
import Board
import Hops
```

2.3.2 Code

Data Type

The `GameState` consists of the current board state and the player whose turn it is next, and, the sequence of hops taken on the previous turn (which will equal the empty list for the starting state). The previous state is also stored, a Maybe type is used for this to handle the case of the beginning of game (when there is no previous state).

```
data GameState = GameState {prevState  :: Maybe GameState,
                            hops       :: [Hop],
                            board      :: Board,
                            turnNext   :: Side} deriving (Show, Eq)
```

Functions

A constant is defined to give the state at the beginning of a new game.

```
startState :: GameState
startState = GameState Nothing [] startBoard Black
```

The successors function takes a GameState and returns a list of all successor GameStates. Once again the List Monad is used as explained previously (§2.2.2, p 10).

```
successors :: GameState → [GameState]
successors gameState@(GameState _ _ board turnNext) =
    do firstHop ← allowedHops board turnNext
       hops     ← continuations firstHop
       let turnNext′ = opposition turnNext
       return $ GameState (Just gameState) hops (newBoard ∘ last $ hops) turnNext′
    where continuations lastHop =
              case continuingHops lastHop of
                []       → [[lastHop]]
                nextHops → do nextHop ← nextHops
                              rest    ← continuations nextHop
                              return  $ lastHop:nextHop:rest
```

The winner function will return Just side value indicating which which side has won the game, or Nothing if neither side has.

```
winner :: GameState → Maybe Side
winner gameState@(GameState _ _ _ turnNext) =
    case successors gameState of
      []        → Just $ opposition turnNext
      otherwise → Nothing
```

Chapter 3

Simple AI Player

3.1 Generic Negamax (Minimax) Algorithm with Alpha-Beta Pruning

In the interests of code reuse Negamax algorithm is separated from the specifics of the game, all that is required is a heuristic function, a successor function, the depth (`ply`), and, the state to be evaluated.

```
module Negamax(negamax) where
```

Negamax can be seen as just a right handed fold over the list of maximum-so-far values of successive negated scores of the successors. The case statement first checks if the `successors` list is empty—indicating the end of the game—and, if it is, returns the heuristic value of the state (which in the circumstances would most probably be not so heuristic). If there are in fact successors a (lazy) list of is generated which each element each element is the maximum of the negated scores of the successors up to that point. This list of partial maximums is then combined by `foldr` which uses the `combine` to force the list up to a value that is greater than β. Because of the lazy semantics of Haskell the successors are only evaluated as far as they need to be—no further.

```
negamax′ :: (a → Int) → (a → [a]) → Int → Int → Int → a → Int
negamax′ heuristic _ 0 _ _ state =
    heuristic state
negamax′ heuristic successorFn ply α β state =
    case successors of
      [] → heuristic state
      otherwise → foldr combine α scores
    where successors = successorFn state
          combine α maxScoreSoFar =
              if α ≥ β
                then β
                else max α maxScoreSoFar
          -- define a (lazy) list of the negations of the algorithm
          -- as applied to all successor states.
          scores = map (uncurry getScore) $ zip successors (scanl max α scores)
          getScore succ α = negate $ negamax′ heuristic successorFn (ply-1) (-β) (-α) succ
```

All that's left is to define the `negamax` function itself, this just passes its arguments in to `negamax′` along with starting α and β cutoffs—maximum and minimum values that an Int (Haskell's fixed size integer type) can represent.

```
negamax :: (a → Int) → (a → [a]) → Int → a → Int
negamax heuristic successorFn ply state = negamax′ heuristic successorFn ply (minBound+1) (maxBound) state
```

3.2 Static Evaluation Functions

A static evaluation function—a heuristic function that is—will be needed. It can be used to make game play decisions on its own but will be more useful later on when combined with the Minimax (or, in this case, Negamax) algorithm (§3.1, p 14).

```
module EvalFuns where
import Board
import GameState
import Negamax
```

A static evaluation function will be defined as a function from a `GameState` (§2.3.2, p 13) to an `Int`. This will allow different evaluation functions to be weighted and combined together. The static evaluation function should always return the "goodness" of the move for the current player.

```
type EvalFun = GameState → Int
```

A function is defined to aid the combination of evaluation functions, it takes a list of `(EvalFun, Int)` tuples and returns an evaluation function which is a linear combination of them.

```
combineEvalFuns :: [(EvalFun, Int)] → EvalFun
combineEvalFuns [] _ = 0
combineEvalFuns ((fn,w):rest) state = ((fn state) ∗ w) + combineEvalFuns rest state
```

A default evaluation function is defined as a combination of the component evaluation functions which are given below..

```
defaultEval = combineEvalFuns [(countCheckers    , 10000),
                               (countKings       , 10000),
                               (closestToKinging , 10),
                               (avgToKinging     , 1),
                               (kingsMoveAway    , 100)]

countCheckers :: EvalFun
countCheckers (GameState _ _ board turnNext) = checkers (opposition turnNext) - checkers turnNext
    where checkers side = length $ allSquaresForSide board side

countKings :: EvalFun
countKings (GameState _ _ board turnNext) = kings (opposition turnNext) - kings turnNext
    where kings side = length $ [()|(_, Just (Checker _ King)) ← allSquaresForSide board side]

closestToKinging  :: EvalFun
closestToKinging gamestate =
    case gamestate of
      (GameState _ _ board Black) → aux (swapSides board)
      (GameState _ _ board White) → aux board
    where aux board = foldl max 0 (rowsForBlack board)

avgToKinging  :: EvalFun
avgToKinging gamestate =
```

```
    case gamestate of
      (GameState _ _ board Black) → aux (swapSides board)
      (GameState _ _ board White) → aux board
    where aux board = if (length (rowsForBlack board)) == 0
                        then 0
                        else (sum (rowsForBlack board)) `div` (length (rowsForBlack board))

-- Helper function
rowsForBlack :: Board → [Int]
rowsForBlack board = [(row loc)|(loc, Just (Checker _ Normal)) ← allSquaresForSide board Black]

kingsMoveAway :: EvalFun
kingsMoveAway gamestate =
    case gamestate of
      (GameState _ _ board Black) → aux board
      (GameState _ _ board White) → aux (swapSides board)
    where aux board = if (length (kingRows board)) == 0
                        then 0
                        else (sum (kingRows board)) `div` (length (kingRows board))
          kingRows board = [(row loc)|
                            (loc, Just (Checker _ King)) ← allSquaresForSide board White]
```

3.3 AI Strategy

```
module Strategies where
import GameState
import EvalFuns(defaultEval, EvalFun)
import Negamax
import Data.List (sortBy)
import Data.Ord (comparing)
```

Once an evaluation function has been built it needs to be combined with Negamax (§3.1, p 14) to create a strategy. The `makeStrategy` function will do this, it takes a static evaluation function and a depth to search and returns a function that will choose the best choice from a list of `GameState`s.

```
makeStrategy :: EvalFun → Int → [GameState] → GameState
makeStrategy eval ply =(λchoices → last $ sortBy compareFn choices)
    where compareFn = comparing $ negamax eval successors ply
```

Finally we can create a strategy!

```
defaultStrategy :: [GameState] → GameState
defaultStrategy = makeStrategy defaultEval 6
```

Chapter 4

ANNs and GAs

4.1 Artificial Neural Network as an Evaluation Function

This module allows the creation of an Artificial Neural Network (ANN) which can be used as an evaluation function for the checkers game. Unfortunately there has not been enough time to do much experimentation with it—or integrate it into the GUI, it can replace the existing evaluation function in the code but there is no GUI widget to do this from the interface—but a reasonably effective set of weights has been found using the Genetic Algorithm (GA) in the `EvolveANN` module (§4.2, p 18).

```
module ANNEvalFun where
import EvalFuns(EvalFun)
import GameState
import Board
import Data.List(foldl′)
import System.Random(mkStdGen, randoms, randomRIO, randomIO)
```

4.1.1 A simple ANN

This module uses multi-layer feed-forward network, the number of layers and the weights of each layer are configurable—and are thus candidates for optimization with the GA.

A type synonym—`ANNWeights`—is defined to represent the weights. It is defined as a list of lists of lists of floating point numbers. The organization is a list for each layer (except the first) containing a list for each neuron in that layer containing a weight for each neuron on the previous layer plus a weight for the bias.

```
type ANNWeights = [[[Float]]]
```

This organization reflects the manner in which each value is needed and makes the code very simple. The code for the ANN itself consists of a single function–`makeANN`–which takes the weights as its first parameter and returns a function from a list of input values—one for each input neuron—to a list of output values. The activation function used is the sigmoid function: $\frac{1}{1+e^{-x}}$.

```
makeANN :: ANNWeights → [Float] → [Float]
makeANN [] inputs = inputs
makeANN (w:ws) inputs = makeANN ws $ map sigmoid weightedSum
    where weightedSum = [foldl′ (+) 0 $ zipWith (∗) w′ (1.0:inputs)| w′ ← w]
          sigmoid x = 1.0 / (1.0 + exp (-x))
```

A function to generate a random set of weights is also provided. It takes a list containing an int for each layer giving the number of neurons in that layer and a seed for the random number generator. Note that this function is completely deterministic, given the same seed it will always return the same weights.

```
randomWeights :: [Int] → Int → ANNWeights
randomWeights (x:x′:xs) seed = weights : randomWeights (x′:xs) seed′
    where gen = mkStdGen seed
          (seed′ : rands) = randoms gen
          weights = [[(j∗2)-1|j ← take (x+1) $ randoms $ mkStdGen k]|k ← take x′ rands]
randomWeights _ _ = []
```

4.1.2 An evaluation function using the ANN

The `annEvalFun` function takes a set of weights and returns an evaluation function—which, as you will remember from the `EvalFuns` (§3.2, p 15) module is itself is a function of `[GameState]` to Int—based on the resulting ANN.

The inputs are created by first swapping over the board if necessary so that the current player is `Black` then making a list with 4 values for each square. There's a value to indicate if the square is occupied by a `White` piece, one to indicate if it's occupied by a `Black` piece and two more to indicate if those pieces are kings.

To get an output the mean of the values of the output neurons is taken and scaled to a number between 0 and the maximum value of `Int`.

```
annEvalFun :: ANNWeights → EvalFun
annEvalFun weights = eval
    where eval gs@(GameState _ _ _ White) = eval gs{board=swapSides $ board gs, turnNext=Black}
          eval (GameState _ _ board _) = let outputs = ann $ inputs board in
                                         round $ (fromIntegral (maxBound :: Int)) ∗
                                                 ((sum $ outputs) / (fromIntegral $ length outputs))
          inputs board = do (_,sc) ← allSquares board
                            -- id is identify so these to lines decide
                            -- whether we′re using the inverse of our
                            -- two functions
                            f1 ← [id, not]
                            f2 ← [id, not]
                            case sc of
                              Nothing → return 0
                              Just (Checker s t) → do if (f1 $ s == White) &&
                                                         (f2 $ t == Normal)
                                                        then return 1
                                                        else return 0
          ann = makeANN weights
```

4.2 Genetic Algorithm to Evolve an ANN

This module defines a very rough and ready Genetic Algorithm (GA) to evolve a set of weights for the ANN defined in the ANNEvalFun module (§4.1, p 17). It uses a tournament style GA, each round a random set of 4 individuals is chosen in two pairs which are played against each other, the two losers are eliminated winners are used as parents for the new individuals that replace them.

Crossover (xover) is implemented by making each sub-tree of the weights come from either a random one of the parents or a combination of the two (recursively). This means that in some cases whole layers will have weights from a single parent but in other cases the random combination will go down the individual weight level. The chance of crossover is currently set at 15%.

Newly created individuals are also mutated, there is a 1 in 300 chance of a given weight being replaced by a new random weight.

The `evolve` function itself takes 2 numbers to indicate the number of hidden and output neurons (a 3 layer ANN is assumed by this module) and the population size to use. It never terminates but instead prints out a winning set of weights after each round, the output can be direct to a file for as long as required then the function terminated manually. GHCi[1] is very useful for experimenting with this module.

```
module EvolveANNN where
import ANNEvalFun
import Strategies
import GameState
import Hops
import Board
import System.Random(mkStdGen, randoms, randomRIO, randomIO)
import Control.Monad(liftM)
import Maybe(fromJust)

evolve :: Int → Int → Int → IO ()
evolve hiddenCount outputCount  popSize =
    evolve′ [(randomWeights [128, hiddenCount, outputCount] i)|i ← [1..popSize]]

evolve′ :: [ANNWeights] → IO ()
evolve′ pop = do
  let popSize = length pop
  i1 ← randomRIO (0,popSize-1)
  i2 ← randomRIO (0,popSize-1)
  let (winnerIdx, loserIdx) = pickWinner i1 i2
  let winner1 = pop !! winnerIdx
  let pop′ = remove loserIdx pop
  i1 ← randomRIO (0,popSize-2)
  i2 ← randomRIO (0,popSize-2)
  let (winnerIdx, loserIdx) = pickWinner i1 i2
  let winner2 = pop′ !! winnerIdx
  let pop′′ = remove loserIdx pop′
  new1 ← breed winner1 winner2
  new2 ← breed winner2 winner1
  putStrLn $ "==========\n\n" ++ (show new1) ++ "\n\n"
  evolve′ $  new1 : new2 : pop′′
   where remove idx lst = (take idx lst) ++ (drop (idx+1) lst)
         -- if neither side wins then White is declared the winner,
         -- since both are picked randomly this pretty much has the
         -- effect of randomly picking a winner.
         pickWinner first second =
             if Just Black == (game (makeStrategy (annEvalFun (pop !! first)) 0)
                                    (makeStrategy (annEvalFun (pop !! second)) 0))
                then (first,  second)
                else (second, first)

game :: ([GameState] → GameState) → ([GameState] → GameState) → Maybe Side
game black white = game′ black white startState
```

[1] A interactive Haskell shell

```
    where game' currentStrategy otherStrategy gameState =
            case successors gameState of
              [] → Just $ opposition $ turnNext gameState -- we have a winner!
              succs → let gameState' = currentStrategy succs in
                       if infiniteLoop gameState' (fromJust $ prevState gameState')
                         then Nothing
                         else game' otherStrategy currentStrategy (currentStrategy succs)
          infiniteLoop a b | (board a) == (board b) = True
          infiniteLoop a (GameState (Just prev) ((SingleHop _ _ _ _) : _) _  _) =
              infiniteLoop a prev
          infiniteLoop _ _ = False

breed :: ANNWeights → ANNWeights → IO ANNWeights
breed parent1 parent2 = do do_xover ← randomRIO (0,100)
                           combined ← if do_xover < 15
                                         then xover parent1 parent2
                                         else return parent1
                           mutated ← mutate combined
                           return mutated

mutate :: ANNWeights → IO ANNWeights
mutate xs = mutate' (mutate' (mutate' mutateWeight)) xs
    where mutate' fn (x:xs) = do x' ← fn x
                                 xs' ← mutate' fn xs
                                 return $ x' : xs'
          mutate' fn [] = return []
          mutateWeight x = do r ← randomRIO (0,300)
                              if r == 1
                                 then liftM(λx → x*2-1) randomIO
                                 else return x

xover :: ANNWeights → ANNWeights → IO ANNWeights
xover [] [] = return []
xover (x:xs) (y:ys) = do r ← randomRIO (0,100)
                         xy ← if r < 30 then
                                   (xover' x y)
                                 else return $ if r < 60 then x else y
                         xys ← xover xs ys
                         return $ xy : xys
    where xover' (x:xs) (y:ys) = do r ← randomRIO (0,100)
                                    let xy = if r < 50 then x else y
                                    xys ← xover' xs ys
                                    return $ xy : xys
          xover' [] [] = return []
```

Chapter 5

GUI

5.1 GUI

The GUI is implemented in GTK[1] using the the gtk2hs[2] binding for Haskell. This is a very low level library requiring a lot of imperative style code (in the IO Monad) and in hindsight I think it might have been better to have used a higher level more functional library.

TODO explain glade TODO explain game state thingies

```
module Main where
import Graphics.UI.Gtk
import Graphics.UI.Gtk.Glade
import Control.Monad
import SimpleGUI(messageBox)

import Board
import GameState
import Hops
import Strategies
```

In order to update the display handles for the various display elements are needed. A data type is used to pass around all the handles as a block rather than try and pass the correct handles to each place that needs them.

```
data GUI = GUI {guiWindow      :: Window,
                guiBoard       :: Table,
                guiPlayerBlack :: CheckButton,
                guiPlayerWhite :: CheckButton}
```

The main loop of the program first calls the gtk2hs `initGUI` function, sets up the widgets, requests the first player move (using the `doTurn` (§5.1, p 22) function), and, enters the GTK main event loop.

```
main :: IO ()
main = do initGUI
          gui ← setupWidgets
          doTurn gui startState
          mainGUI
```

[1] http://www.gtk.org/

[2] http://www.haskell.org/gtk2hs/

The `setupWidgets` function gets references to the widgets which are defined in the `.glade` XML file and sets up stuff which isn't (such as the board grid). It returns a `GUI` data structure (§5.1, p 21) containing all the references which will be needed by the rest of the program.

```
setupWidgets :: IO GUI
setupWidgets = do Just xml   ← xmlNew "checkers.glade"
                  window     ← xmlGetWidget xml castToWindow "gameWindow"
                  container  ← xmlGetWidget xml castToAlignment "container"
                  [playerBlack, playerWhite] ← mapM (xmlGetWidget xml castToCheckButton)
                                                ["playerBlack", "playerWhite"]
                  -- Now recreate the board, twice as many cells as
                  -- there are squares on the board because we want to
                  -- divide squares into 4 for the arrows
                  table ← tableNew 16 16 True
                  containerAdd container table
                  onDestroy window mainQuit
                  widgetShowAll window
                  return $ GUI window table playerBlack playerWhite
```

The `doTurn` function checks the current status of the player type check boxes to determine if AI player should be used or if the human should be asked. This means the type of player can be changed mid-game. It also is responsible for checking if a winner has been found and will display a message box if one has.

```
doTurn :: GUI → GameState → IO ()
doTurn gui state =
    case winner state of
      Just w → do let w′ = show w
                  playerTurn gui state -- Causes the board display to
                                       -- be updated
                  messageBox (w′ ++ " wins!") $
                             "We have a winner!\n" ++
                             "Congratulations to " ++ w′ ++ "!"
                  -- reset the board
                  doTurn gui startState
      Nothing → do human ← toggleButtonGetActive toggle
                   case human of
                     True  → playerTurn gui state
                     False → aiTurn gui state
    where toggle = case turnNext state of
                     White → guiPlayerWhite gui
                     Black → guiPlayerBlack gui
```

The `aiTurn` function first hands calculates the AI move using `defaultStrategy` (§3.3, p 16). Once it has the move it intends to make it calls `showTurn` which will show each hop of the move with a small gap in between—this will allow the user to see the move being made. Because `timeoutAdd` is used the function will return after the first hop is shown but will be scheduled to run again by the main event loop.

```
aiTurn :: GUI → GameState → IO ()
aiTurn gui state = let state′@(GameState _ hops _ _) = (defaultStrategy $ successors state) in
                  showTurn state′ [] hops
        where showTurn state hops (h:hs) =
                  do clearBoard gui
                     drawHops gui (h:hops)
                     drawBoard gui (newBoard h)
                     widgetShowAll $ guiWindow gui
                     timeoutAdd (showTurn state (h:hops) hs >> return False) 300
                     return ()
```

```
            showTurn state _ [] = doTurn gui state
```

The `clearButton` function is used by both `playerTurn` and `aiTurn` to clear the board of all widgets before recreating it using the new game state.

```
clearBoard :: GUI → IO ()
clearBoard gui = do pieces ← containerGetChildren (guiBoard gui)
                    mapM_ (containerRemove (guiBoard gui)) pieces
```

`drawHops` will draw an indication of the last move made. This enables the player to see where the AI moved last turn.

```
drawHops :: GUI → [Hop] → IO ()
drawHops gui hops = mapM_ drawHop hops
    where drawHop (SingleHop _ _ dest _)  = overlay dest "images/marker.png"
          drawHop (CaptureHop _ _ cap dest _)  = do overlay dest "images/marker.png"
                                                    overlay cap "images/capture.png"
          overlay loc file = do image ← imageNewFromFile file
                                tableAttach (guiBoard gui)
                                            image
                                            (col loc * 2)
                                            (col loc * 2 + 2)
                                            (row loc * 2)
                                            (row loc * 2 + 2)
                                            [Shrink] [Shrink] 0 0
                                return ()
```

`drawBoard` draws the actual squares onto the checker board. It simply calls `drawSquare` for each square on the board, this loads the correct image (given by `squareImage`) and attaches it into the board table at its location.

```
drawBoard :: GUI → Board → IO ()
drawBoard gui board = mapM_ (uncurry drawSquare) (allSquares board)
  where drawSquare loc square = do
          image ← imageNewFromFile (squareImage square)
          tableAttach (guiBoard gui)
                      image
                      (col loc * 2)
                      (col loc * 2 + 2)
                      (row loc * 2)
                      (row loc * 2 + 2)
                      [Shrink] [Shrink] 0 0
        squareImage Nothing                      = "images/empty.png"
        squareImage (Just (Checker White Normal)) = "images/white.png"
        squareImage (Just (Checker White King))   = "images/white_king.png"
        squareImage (Just (Checker Black Normal)) = "images/black.png"
        squareImage (Just (Checker Black King))   = "images/black_king.png"
```

`playTurn` is responsible for drawing the current board state to the window and allowing the user to select their next move. The `drawChoices` function (§5.1, p 24) will draw the arrow buttons for each checker and direction that can be moved. Once the user has selected a hop the callback `doHop'` will be called which will check if the move is complete and either move to the next turn or loop round and ask for the next hop.

```
playerTurn :: GUI → GameState → IO ()
playerTurn gui state@(GameState prevState hops board0 side) =
    doHop board0 $ allowedHops board0 side
    where succs = successors state
```

```
doHop board choices = do
  clearBoard gui
  drawChoices gui choices (doHop′ choices)
  drawHops gui hops
  drawBoard gui board
  widgetShowAll $ guiWindow gui
doHop′ choices hop = let b = (newBoard hop) in
                     case filter ((==b) ∘ board) succs of
                       (s : _) → doTurn gui s
                       otherwise  →  doHop b $ continuingHops hop
```

drawChoices is the method by which playerTurn prompts the user for his/her next move. It displays up to 4 buttons on the corners of each check that can be moved, the click handler for each is set to a closure which will call the callback function passing on the choice made.

```
drawChoices :: GUI → [Hop] → (Hop → IO ()) → IO ()
drawChoices gui choices callback = mapM_ showChoice choices
    where showChoice hop = do
            let (src,dst) = (source hop, destination hop)
            let tablerow = (row src ∗ 2) + if row dst > row src then 1 else 0
            let tablecol = (col src ∗ 2) + if col dst > col src then 1 else 0
            image ← imageNewFromFile (if row dst > row src
                                        then if col dst > col src
                                               then "images/arrowdr.png"
                                               else "images/arrowdl.png"
                                         else if col dst > col src
                                               then "images/arrowur.png"
                                               else "images/arrowul.png")
            button ← buttonNew
            onButtonPress button $(λ_ → callback hop >> return True)
            buttonSetImage button image
            buttonSetRelief button ReliefNone
            tableAttach (guiBoard gui)
                        button
                        tablecol
                        (tablecol + 1)
                        tablerow
                        (tablerow + 1)
                        [Fill] [Fill] 0 0
```

Index