COMS 4995 Parallel Functional Programming
Project Report : **A.I. Hangman**

Anthony Pitts
{UNI: aep2195}

Fall 2021

Contents

1 Project Abstract

For my Parallel Functional Programming final project, I have implemented a version of the game "Hangman" in Haskell. My implementation uses AI heuristics and parallel strategies to predict the next best letter choice in each round of the game. This involved a significant amount of dictionary/word processing, AI prediction heuristics, and Monadic operations that lent themselves very well to parallelism.

2 Hangman Version Explanation

At its core, Hangman is a multi-round game in which the player (or AI, in this case) continuously guesses letters of an unknown word before they run our of guesses.

Unlike the classic version of Hangman, however, my implementation did not restrict the number of attempts to select a letter to the 5 or so "body parts" of the hangman. Instead, the program runs until the correct word has been discovered by the AI. The reason behind this difference is to demonstrate that the program is actually working towards the goal of finding the word, and it is not simply making 5 incorrect guesses to reduce the runtime. This way, the better the algorithm and use of parallelism, the faster the program will solve the word puzzle.

3 Sequential Implementation and Performance

3.1 Program Steps

The logic and control flow of the sequential algorithm that was implemented before introducing parallelism as follows:

1. From the command-line arguments, gather the dictionary file path, the number of letters that the words being guessed must have, and the number of Hangman games to simulate.

2. Read in the dictionary, filtering words with different lengths than the user-specified length.

3. For each game, select a random word from the dictionary for the AI to guess.

4. Continuously call the `selectLetter` "function" each round until the AI has guessed every letter in the word.

5. Repeat steps 3-4 until N games have been simulated, where N was specified by the user's command-line argument.

3.2 AI Letter Selection Heuristics

Each round of the game, the program must select a letter that it believes could be a missing letter in the word being guessed. My algorithm uses the heuristic of the "most common" possible letter. In other words, it considers all words that have the same length as the word being guessed, and those with letters in the same positions as the correct letters already guessed. Of these words, it selects the letter that appears most frequently amongst them (excluding letters already guessed).

Since this is very computationally heavy to run on the dictionary each round, I implemented the following two pruning heuristics:

1. **Correct Letter Pruning:** If the program guessed the correct letter, update the dictionary of words by removing words that do not match the updated letter positions with the newly discovered letter incorporated.

```haskell
-- Update dictionary of words if this correct letter was guessed
--   by removing words that don't match the updated letter positions
pruneCorrect :: InternalState -> [Types.Word]
pruneCorrect is = filter positionAgreement dictWords
    where dictWords = wordDict is
          validLetter maybeLetter dictWordLetter = case maybeLetter of
              (Just l) -> dictWordLetter == l
              _ -> True
          positionAgreement w = and $ zipWith validLetter (currentWord is) (getWordString w)
```

2. **Incorrect Letter Pruning:** If the program guessed the incorrect letter, update the dictionary of words by removing all words that contain the incorrectly guessed letter.

```haskell
-- Update dictionary of words if this incorrect letter was guessed
--   by removing words that have the incorrect letter
pruneIncorrect :: InternalState -> Char -> [Types.Word]
pruneIncorrect is c = filter wordsWithLetter possibleWords
    where possibleWords = wordDict is
          wordsWithLetter w = not $ elem c (getWordString w)
```

3.3 Data Types

Central to the simple syntax and efficient semantics of the program was the encapsulation of certain pieces of data into the following data types.

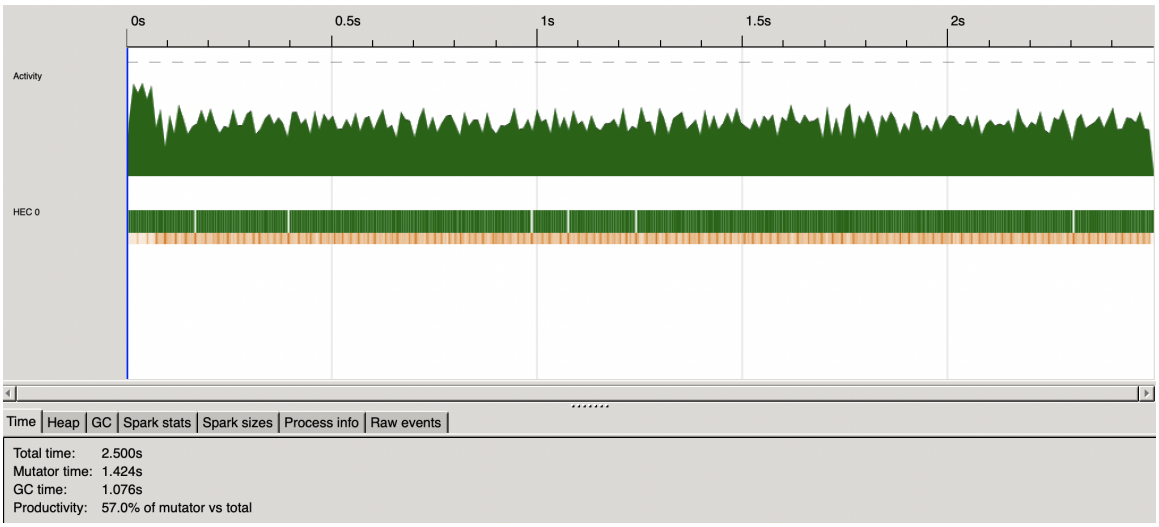
1.
```haskell
data InternalState = InternalState {
  wordDict :: [Types.Word],
  wordToGuess :: String,
  usedLetters :: Set Char,
  currentWord :: [Maybe Char]
} deriving (Show)
```
The `InternalState` data structure is used to hold critical information on the state of the Hangman game including the valid words left (`wordDict`), the word being guessed (`wordToGuess`), the letters already guessed (`usedLetters`), and the `currentWord` with the correctly guessed letters resolved to `Just Char`.

2.
```haskell
data HangmanGameF a =
    PlayerTurn InternalState (Guess -> a)
  | GameOver TerminalState
instance Functor HangmanGameF where
  fmap f (PlayerTurn is g) = PlayerTurn is (f . g)
  fmap _ (GameOver ts)     = GameOver ts
```
The `HangmanGameF` data structure is used to specify the current state of the game as either the Player's turn or `GameOver`. Whereas `GameOver` results in a `TerminalState` that ends the game, the `PlayerTurn` type carries the game's `InternalState` and a function that takes the AI's next letter guess to proceed to the next game state. To control the flow of the game from one guess/round to the next, the `HangmanGameF` type implements `Functor`.

3.4 Sequential Performance

Unless otherwise stated, all Threadscope performance statistics shown in this report were run on the same dictionary, number of simulated games, length of words being guessed, and cores. This is on the large dictionary provided in the submission folder, words of length 7, simulating 50 games, and 2 cores (except for the sequential program statistics below, which is running on 1 core). Keeping these inputs static allows for a fair and controlled assessment of performance as parallelism gets introduced.

The overall performance of the sequential implementation is summarized by Threadscope as follows:



Aside from the glaring runtime of 2.5 seconds, the health of the program is relatively balanced. There is a spike in activity at the very start of the program, which is likely due to the process of reading in the dictionary. Other than that, the intensity of activity remains relatively high throughout the program's run, yet does not have any random, extreme spikes. This will prove to be relevant in the selection of the best parallel strategies later.

Ultimately, the goal of parallelizing this program was to reduce the runtime as much as possible, since that is the primary problem with the usability of this Hangman predictor.

4 First Attempts at Parallelizing Hangman

4.1 Creating an NFData Instance for Word Data Type

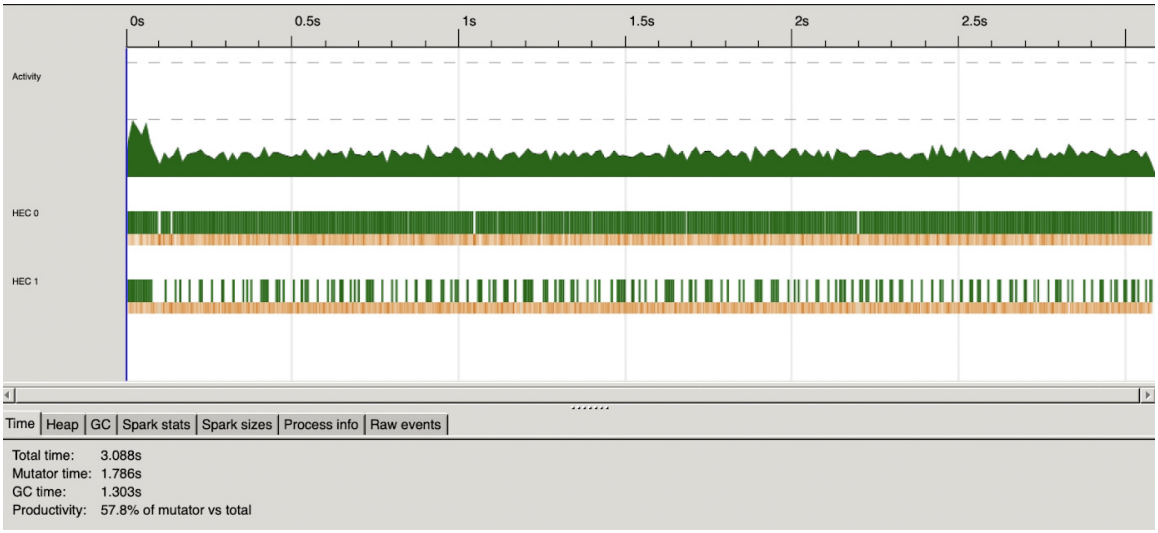
After a significant amount of experimentation with several parallel strategies, I found the most valuable for this program to be `rdeepseq`. This is largely because the parallel processes handle words and lists of words, represented as `[Word String]`. When the program is resolving a `Word`, it is useful to resolve it to Normal Form, including the underlying `String` it holds. However, in order to take advantage of `rdeepseq` on the `Word` data type, I had to make an instance of `NFData` for `Word`. The code for creating this instance is as follows:

```
newtype Word = Word String deriving Show
instance NFData Types.Word where
  rnf (Word s) = Word s `seq` ()
```

4.2 Parallel Dictionary Processing

As previously pointed out, the Threadscope analysis of the sequential implementation showed a spike in CPU activity at the very start of the program. This was likely the result of reading the dictionary into a list of `Word`s.

In order to distribute this work onto multiple cores, I used a simple chucking strategy to group the words being processed and realized the following change in runtime:

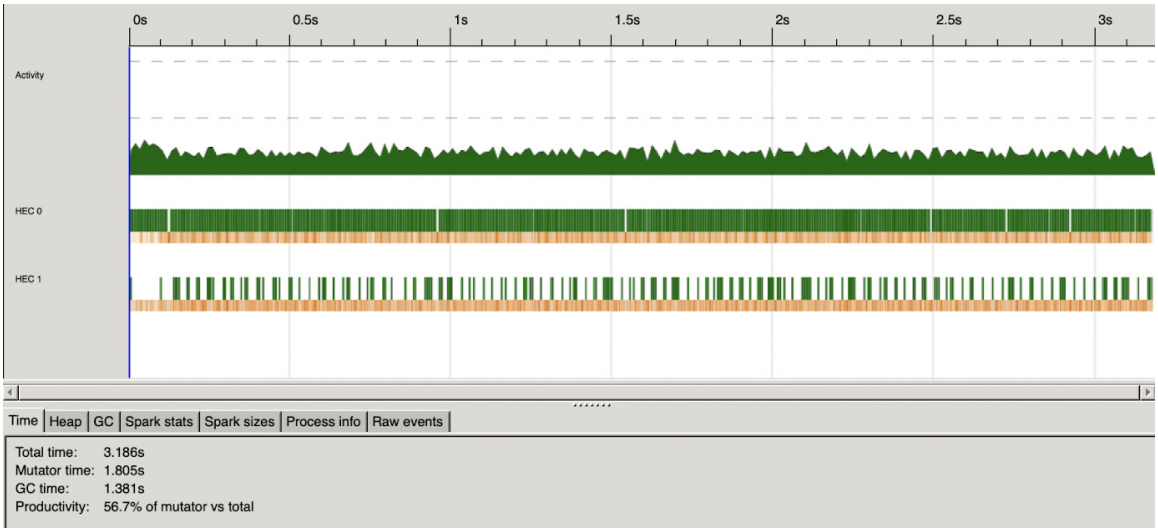


In retrospect, the value of this change was very minor and ultimately unnecessary since the size of a dictionary is quite small. Despite the fact that both cores were converting sparks during the dictionary reading, it was sometimes to the detriment of the runtime because the overhead of generating all the sparks outweighed any speed-up realized from parallelization.

4.3 Parallel Pruning Strategies

The other aspect of the program that I made parallel, and later realized its uselessness, was using `parMap rdeepseq` on the words being pruned (for both correct and incorrect pruning). I tried using many different sizes of chunks to parallelize the process, yet saw no gain on any size chunk.

Ultimately, I found that the problem was that the work being done for each word was very little and the first core was able to complete each "chunk" of words quickly, leaving most sparks to get garbage collected. As the below Threadscope report shows, there was a high density of garbage collected sparks and an increased runtime from the sequential implementation:



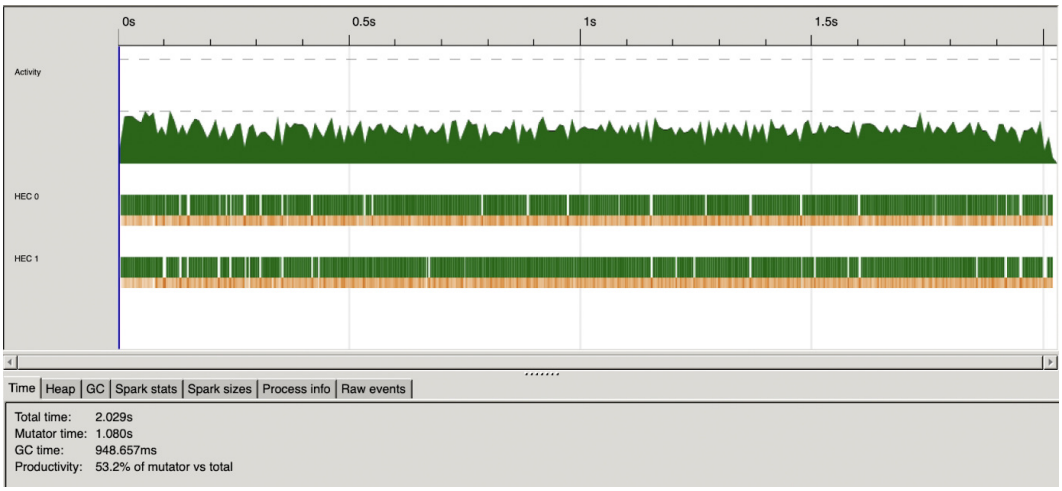
5 Successful Parallelization Strategies

5.1 Parallelizing Game Simulations

The first, successful parallel addition to the program was running multiple Hangman games at once. In order to achieve this, I took advantage of `parMap rdeepseq` by using that strategy to call the `setupGame` function on all the words being guessed (1 word per game).

```haskell
-- Simulate several Hangman games.
simulateGames :: String -> Int -> Int -> IO [String]
simulateGames dictFile wordsLength numberOfGames = do
  validWords <- loadWordList wordsLength dictFile
  wordsToGuess <- getRandomWords validWords numberOfGames
  return $ force (parMap rdeepseq (setupGame validWords) wordsToGuess)
```

By sparking the call to each game simulation, I was able to appreciate a significant gain in runtime as shown below:



From this Threadscope analysis, it is also clear that, unlike in the past 2 attempts at utilizing multiple cores, this run showed a significant amount of sparks being converted by both the first and second cores (as shown in green in the diagram).

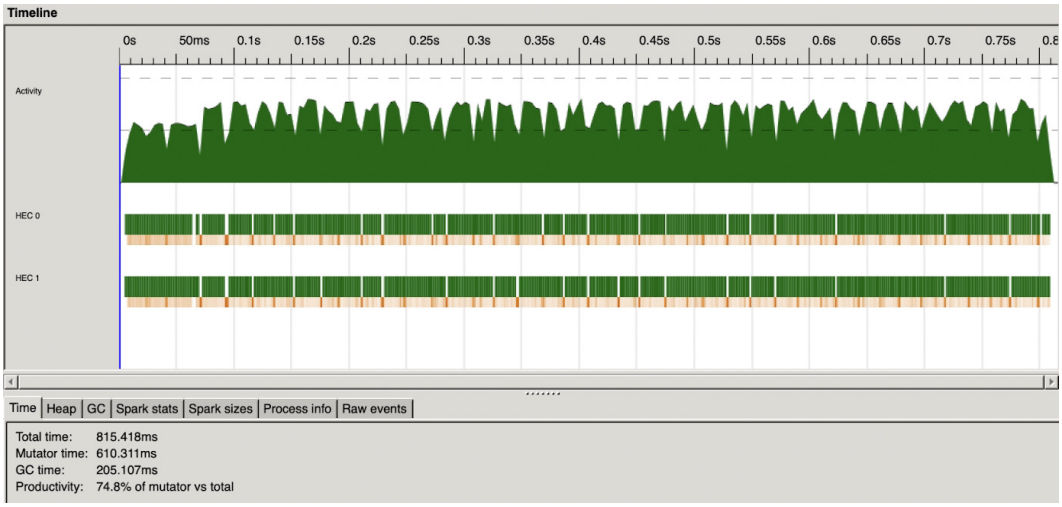
With these changes, the runtime was reduced to 2.03 seconds. This is a **16.4% decrease in runtime** from the sequential implementation.

5.2 Parallelizing Letter Selection

The other, successful parallel addition to the program was around generating the letter frequencies amongst all possible words to determine the "most common" letter. I did this by breaking up the list of words to be processed into chunks whose sizes were dynamically determined as a function of the number of letters guessed. This way, the more letters that have been guessed, the smaller the size of each individual "chunk." Again, I utilized `parMap rdeepseq` by using that strategy to call the `updateWordsLetterCounts` function that builds a map of `Char` to `Int`, representing the letter frequencies.

```
generateLetterFreqs :: Set Char -> [Types.Word] -> Map Char Int
generateLetterFreqs guessedLetters wordList = foldr joinMaps emptyLetterMap mapsFromChunks
  where mapsFromChunks :: [Map Char Int]
        mapsFromChunks = parMap rdeepseq (updateWordsLetterCounts guessedLetters emptyLetterMap) (chunksOf chunkSize wordList)
        chunkSize = 60 `div` (size guessedLetters + 1)
        joinMaps m1 m2 = unionWith (+) m1 m2
```

The use of this parallel paradigm had the following impact on Hangman's Threadscope report, as shown below:



From this Threadscope analysis, one can notice a few important aspects:

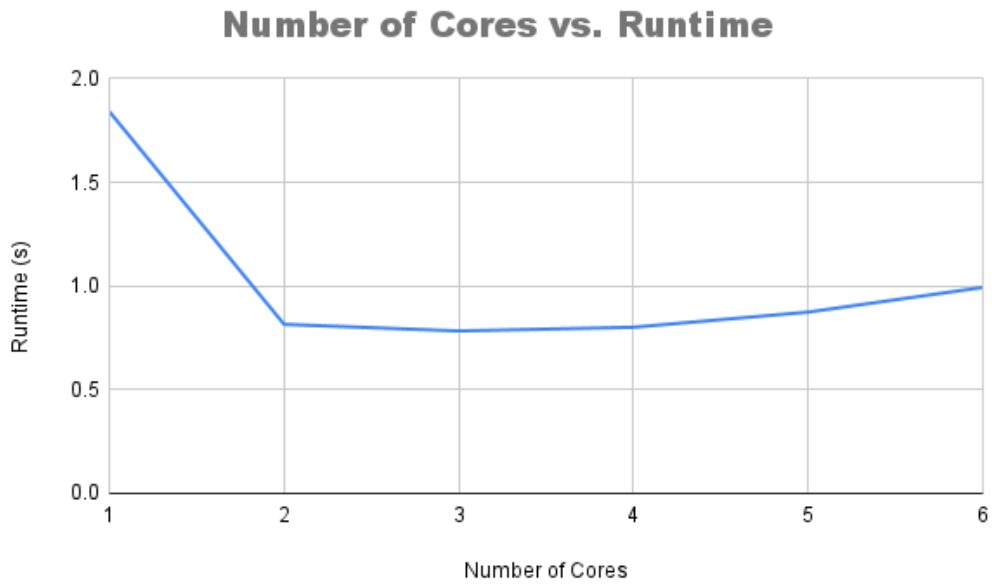
1. The general activity monitor shows that the CPU is far better balanced than previously realized. The work on the CPU is relatively constant and shows no moments of exhaustive computation.

2. The percent of sparks being garbage collected is at its lowest, with only 205ms dedicated to garbage collection.

3. With these changes, the runtime was reduced to 815 milliseconds. This is a **67.4% decrease in runtime** from the sequential implementation. In other words, the complete parallel implementation runs **3-4 times as fast as the sequential implementation**.

5.3 How Multiple Cores Impacted Runtime

After completion of all the parallel components of the Hangman program, I began to investigate the impact that the number of cores had on the overall runtime of the program. The below graph summarizes my findings:



Running on a computer with only 2 cores, yet 4 threads, the above results confirm the notion that my computer is only able to perform productive, parallel work on those 2 cores. Hence, within the constraints of my hardware, this program runs much faster on 2 cores than 1. Yet, telling it to run on more than 2 proved to be pointless. Running on 3 or more cores resulted in approximately the same runtime, if not slightly worse than 2 cores.

As expected, the average runtime on 1 core is slightly less than a 2x speedup from the sequential implementation. This was nearly ideal since 1 core on my computer has 2 threads whose absolute best speedup would have been 2x. In the same vein, the average runtime on 2 cores was a 3-4x speedup from the sequential program. Again, this was nearly ideal since 2 cores on my computer has 4 threads, whose absolute best speedup would have been 4x.

6 Test Suite

Details on how to build, run, and test the program can be found in the README file, located at the root of the code submission.

This program comes with a full test suite which confirms the success of the program in both sequential and parallel mode. Also, it performs many unit tests that are generally there for sanity checking certain components such as the expectation of the relative runtimes if the Hangman games are simulated on a large dictionary or small one, if the AI is guessing long vs short words, or if the number of games being simulated is large or small.

7 Conclusion and Key Takeaways

My experience coding this parallel program taught me many important lessons about functional and parallel programming. One such lesson is to find where the bulk of effort is being

done in the program and parallelize that first. One problem I ran into was that I tried parallelizing the smaller tasks first, such as reading in the dictionary, before parallelizing the bigger tasks, such as generating the letter frequencies.

I would also generalize this notion to another lesson, which is to not attempt to parallelize all the work. When I tried to refactor my code to use parallel strategies on minor components, I found that the overhead of garbage collection became too significant. Often, I realized that this overhead was not worth it for a small piece of computation that could just be done sequentially.

Lastly, the use of The Haskell Tool Stack for this program showed me a side of Haskell that I did not get from completing the homework assignments. Seeing how to take many different Haskell files and build up a full application showed me how Haskell can scale to industry-level software.

8 Program Source Code

8.1 Runner.hs

```haskell
{-
The Runner module is responsible for simulating numerous games of Hangman
and the general control flow of each game.
-}

module Runner where

import          Control.Monad.Free
import          Hangman
import          Types
import          LetterGuesser
import          DictProcessor
import          Control.Parallel.Strategies(rdeepseq, parMap)
import          Control.DeepSeq(force)

-- Simulate several Hangman games.
simulateGames :: String -> Int -> Int -> IO [String]
simulateGames dictFile wordsLength numberOfGames = do
  validWords <- loadWordList wordsLength dictFile
  wordsToGuess <- getRandomWords validWords numberOfGames
  return $ force (parMap rdeepseq (setupGame validWords) wordsToGuess)

-- Initialize Hangman game state
setupGame :: [Types.Word] -> Types.Word -> String
setupGame wordList wordBeingGuessed = runGame $ playGame state
  where state = Hangman.createInitialState wordList wordBeingGuessed

-- Run a sinlge game of Hangman
runGame :: HangmanGame a -> String
runGame (Free (PlayerTurn is next)) = do
  let letterGuess = Guess $ selectLetter is
  runGame $ next letterGuess
```

```
runGame (Free (GameOver (Win is))) = wordToGuess is
runGame _ = error "Error: Game lost. This should never happen."
```

8.2 DictProcesssor.hs

```
{-
The DictProcessor module is responsible for processing
the dictionary that valid words are derived from.
-}

module DictProcessor where

import          Hangman(isPlayableLetter)
import          Data.Maybe
import          Data.Char
import          Types
import          System.Random (randomRIO)
import          Data.List.Split(chunksOf)
import          Control.Parallel.Strategies(rdeepseq, parMap)

-- Load the dictionary of words
loadWordList :: Int -> FilePath -> IO [Types.Word]
loadWordList wordsLength = fmap processWords . readFile
  where processWords fileStr = concat $ parMap rdeepseq maybeWords (chunksOf 100
      (lines fileStr))
       word w
         | length w == wordsLength && (and $ map isPlayableLetter w) = Just (Word $
             map toLower w)
         | otherwise = Nothing
       maybeWords wordList = mapMaybe word wordList

-- Select random words from the dictionary of words
getRandomWords :: [Types.Word] -> Int -> IO [Types.Word]
getRandomWords dict n = do indices <- randomDictIndices n (length dict)
                           return $ collectWords dict 0 indices
    where collectWords [] _ _ = []
          collectWords (x:xs) index indices
            | elem index indices = getWord index indices x ++ collectWords xs (succ
                index) indices
            | otherwise = collectWords xs (succ index) indices
          getWord _ [] _ = []
          getWord a (x:xs) p
            | a == x = p : getWord a xs p
            | otherwise = getWord a xs p

-- Generates a random set of indicies in the list of valid words to determine
--   which words to use in each Hangman game being simulated.
randomDictIndices :: Int -> Int -> IO [Int]
randomDictIndices n dictLength = do
  randomIndex <- randomRIO (0, dictLength - 1)
```

```
  case n <= 1 of
    True -> return [randomIndex]
    _ -> do randomIndices <- randomDictIndices (pred n) dictLength
            return $ randomIndex : randomIndices
```

8.3 LetterGuesser.hs

```
{-
The LetterGuesser module is responsible for the AI process of selecting
the next letter to be guessed each round of Hangman. LetterGuesser uses
two pruning strategies depending on whether a guessed letter was in the
word being guessed or not.
-}

module LetterGuesser where

import          Control.Monad(liftM2)
import          Types
import          Data.Map (Map, fromList, insertWith, lookup, unionWith)
import          Data.Set (Set, member, difference, fromList, toList)
import          Data.List.Split(chunksOf)
import          Control.Parallel.Strategies(parMap, rdeepseq)


-- Update dictionary of words if this correct letter was guessed
--   by removing words that don't match the updated letter positions
pruneCorrect :: InternalState -> [Types.Word]
pruneCorrect is = concat $ parMap rdeepseq filterChunks (chunksOf 1000 dictWords)
    where filterChunks wordChunk = filter positionAgreement wordChunk
          dictWords = wordDict is
          validLetter maybeLetter dictWordLetter = case maybeLetter of
              (Just l) -> dictWordLetter == l
              _ -> True
          positionAgreement w = and $ zipWith validLetter (currentWord is)
               (getWordString w)

-- Update dictionary of words if this incorrect letter was guessed
--   by removing words that have the incorrect letter
pruneIncorrect :: InternalState -> Char -> [Types.Word]
pruneIncorrect is c = concat $ parMap rdeepseq filterChunks (chunksOf 1000
    possibleWords)
    where filterChunks wordChunk = filter wordsWithLetter wordChunk
          possibleWords = wordDict is
          wordsWithLetter w = not $ elem c (getWordString w)

-- Top-level function that uses AI strategies to determine the
--   best letter to guess given the game's InteralState
selectLetter :: InternalState -> Char
selectLetter is = mostCommonLetter letterFrequencies unguessedLetters
    unguessedLettersList (head unguessedLettersList)
```

```
    where possibleWords = wordDict is
          guessedLetters = usedLetters is
          unguessedLetters = difference (Data.Set.fromList alphabet) guessedLetters
          unguessedLettersList = toList $ difference (Data.Set.fromList alphabet)
              guessedLetters
          letterFrequencies = generateLetterFreqs guessedLetters possibleWords

generateLetterFreqs :: Set Char -> [Types.Word] -> Map Char Int
generateLetterFreqs guessedLetters wordList = foldr joinMaps emptyLetterMap
    mapsFromChunks
  where mapsFromChunks :: [Map Char Int]
        mapsFromChunks = parMap rdeepseq (updateLetterCountsWithWords
            guessedLetters emptyLetterMap) (chunksOf 60 wordList)
        joinMaps m1 m2 = unionWith (+) m1 m2

-- Selects the most common letter from the given letter frequency map
mostCommonLetter :: Map Char Int -> Set Char -> [Char] -> Char -> Char
mostCommonLetter _ _ [] mostFreqLetter = mostFreqLetter
mostCommonLetter letterFreqs unguessedLetters (currentLetter:xs) mostFreqLetter =
    case liftM2 (>) (freqCount mostFreqLetter) (freqCount currentLetter) of
      Just (True) -> mostCommonLetter letterFreqs unguessedLetters xs mostFreqLetter
      _ -> mostCommonLetter letterFreqs unguessedLetters xs currentLetter
    where freqCount l = Data.Map.lookup l letterFreqs

-- Update the letter frequency map with the letters in the given word chunk (list)
updateLetterCountsWithWords :: Set Char -> Map Char Int -> [Types.Word] -> Map Char
    Int
updateLetterCountsWithWords guessedLetters m wordList = foldr (updateLetterCounts
    guessedLetters) m wordList

-- Update the letter frequency map with the letters in the given word
updateLetterCounts :: Set Char -> Types.Word -> Map Char Int -> Map Char Int
updateLetterCounts guessedLetters (Types.Word w) m = foldr (incLetterCount
    guessedLetters) m w

-- Increment the frequency of the given letter in the letter frequency map
incLetterCount :: Set Char -> Char -> Map Char Int -> Map Char Int
incLetterCount guessedLetters c m
    | member c guessedLetters = m
    | otherwise = insertWith (+) c 1 m

emptyLetterMap :: Map Char Int
emptyLetterMap = Data.Map.fromList $ map (\l -> (l, 0)) alphabet
```

8.4 Types.hs

```
{-
The Types module is responsible for defining the many data types used
in the Hangman program.
-}
```

```haskell

module Types where

import          Data.Set(Set, fromList)
import          Control.Monad.Free
import          Control.DeepSeq(NFData, rnf)

data InternalState = InternalState {
  wordDict :: [Types.Word], -- dynamic list of words that gets smaller as the
      program makes guesses
  wordToGuess :: String,
  usedLetters :: Set Char,
  currentWord :: [Maybe Char]
} deriving (Show)

-- Types that determine the current state of the Hangman game
newtype RunningState = RunningState InternalState deriving (Show)
data TerminalState = Win InternalState | Loss InternalState deriving (Show)
data GameState = Terminal TerminalState | Running RunningState deriving (Show)

-- Free Monad to abstract the game loop
data HangmanGameF a =
    PlayerTurn InternalState (Guess -> a)
  | GameOver TerminalState
instance Functor HangmanGameF where
  fmap f (PlayerTurn is g) = PlayerTurn is (f . g)
  fmap _ (GameOver ts)   = GameOver ts
type HangmanGame a = Free HangmanGameF a

newtype Guess = Guess Char

newtype Word = Word String deriving Show
instance NFData Types.Word where
  rnf (Word s) = Word s `seq` ()

-- Gets the underlying strin of the Word type
getWordString :: Types.Word -> String
getWordString (Word s) = s

alphabet :: [Char]
alphabet = ['a'..'z']

alphabetSet :: Set Char
alphabetSet = fromList ['a'..'z']
```

8.5 Hangman.hs

```haskell
{-
The Hangman module is responsible for controling the internal game
state. Thus, once a letter is guessed, this module validates
```

```haskell
the guess and updates the game's interal state with that guess.
-}

module Hangman where

import          Control.Monad.Free
import          Data.Char
import          Data.Maybe
import          Data.Set (empty, insert, member)
import          Types
import          LetterGuesser(pruneCorrect, pruneIncorrect)

-- Get the internal state from RunningState
internalState :: RunningState -> InternalState
internalState (RunningState is) = is

isPlayableLetter :: Char -> Bool
isPlayableLetter c = member (toLower c) alphabetSet

-- Initialize the state with the word being guessed and subset of
--  the dictionary with words of equal length to the word being guessed
createInitialState :: [Types.Word] -> Types.Word -> RunningState
createInitialState wordDictionary (Word w) = RunningState InternalState {
  wordDict = wordDictionary,
  wordToGuess = w,
  usedLetters = empty,
  currentWord = map (\_ -> Nothing) w
}

-- Update the RunningState with a letter Guess
applyGuess :: RunningState -> Guess -> GameState
applyGuess (RunningState is) g
    | all isJust (currentWord newIs) = Terminal $ Win newIs
    | otherwise = Running $ RunningState newIs
  where newIs = updateState is g

-- Update the InternalState with the new Guess
updateState :: InternalState -> Guess -> InternalState
updateState is (Guess c)
  | c `member` usedLetters is = is
  | any (eqIgnoreCase c) (wordToGuess is) = is {
    wordDict = pruneCorrect is,
    usedLetters = insert c (usedLetters is),
    currentWord = zipWith (getCharMaybe c) (wordToGuess is) (currentWord is)
  }
  | otherwise = is {
    wordDict = pruneIncorrect is c,
    usedLetters = insert c (usedLetters is)
  }
  where
    eqIgnoreCase char c' = char == toLower c'
```

```haskell
    -- getCharMaybe returns Maybe of the letter if the guess was the letter, else
        Nothing
    getCharMaybe _ _ (Just x) = Just x
    getCharMaybe guessedChar char _
      | guessedChar == toLower char = Just char
      | otherwise = Nothing

gameOver :: TerminalState -> HangmanGame ()
gameOver ts = liftF $ GameOver ts

playerTurn :: InternalState -> HangmanGame Guess
playerTurn is = liftF $ PlayerTurn is id

-- Hangman Game loop. Ends once it is in a Terminal state.
playGame :: RunningState -> HangmanGame ()
playGame rs = do
  c <- playerTurn (internalState rs)
  case applyGuess rs c of
    Running rs' -> playGame rs'
    Terminal ts -> gameOver ts
```

8.6 Main.hs

```haskell
module Main where

import Runner
import System.Environment(getArgs, getProgName)
import System.Exit(die)
import Text.Read(readMaybe)

main :: IO ()
main = do
   args <- getArgs
   case args of
      [filename, wordsLength, numberOfGames] -> do
         let getWordsLength = case readMaybe wordsLength of
               Just l -> l
               _ -> error "wordsLength must be an integer"
             getNumberOfGames = case readMaybe numberOfGames of
               Just l -> l
               _ -> error "numberOfGames must be an integer"
         results <- simulateGames filename getWordsLength getNumberOfGames
         putStrLn $ "Solved " ++ (show $ length results) ++ " hangman games."
         return ()
      _ -> do
           pn <- getProgName
           die $ "Usage: " ++ pn ++ " <filename> <wordsLength>"
```

8.7 RunTests.hs

```haskell
module Main where

import Runner
import Test.HUnit
import System.CPUTime

main :: IO Counts
main = runTestTT $ TestList [
  TestLabel "testDictSizes" testDictSizes,
  TestLabel "testWordSizes" testWordSizes,
  TestLabel "testNumberGames" testNumberGames
  ]

testDictSizes :: Test
testDictSizes = TestCase $ do
  -- large dictionary test
  largeStartTime <- getCPUTime
  largeDictResults <- simulateGames "data/large_dict.txt" 8 50
  largeDictResults `seq` return ()
  largeEndTime <- getCPUTime
  let largeDictTime = (largeEndTime - largeStartTime)

  -- medium dictionary test
  mediumStartTime <- getCPUTime
  mediumDictResults <- simulateGames "data/medium_dict.txt" 8 50
  mediumDictResults `seq` return ()
  mediumEndTime <- getCPUTime
  let mediumDictTime = (mediumEndTime - mediumStartTime)

  -- small dictionary test
  smallStartTime <- getCPUTime
  smallDictResults <- simulateGames "data/small_dict.txt" 8 50
  smallDictResults `seq` return ()
  smallEndTime <- getCPUTime
  let smallDictTime = (smallEndTime - smallStartTime)

  let testResult = smallDictTime < mediumDictTime && (mediumDictTime <
      largeDictTime)
  assertBool "Smaller dictionaries should run faster than larger dictionaries!"
      testResult

testWordSizes :: Test
testWordSizes = TestCase $ do
  -- large word lengths test
  largeStartTime <- getCPUTime
  largeWordsResults <- simulateGames "data/medium_dict.txt" 8 50
  largeWordsResults `seq` return ()
  largeEndTime <- getCPUTime
  let largeWordsTime = (largeEndTime - largeStartTime)
```

```
48
49    -- medium word lengths test
50    mediumStartTime <- getCPUTime
51    mediumWordsResults <- simulateGames "data/medium_dict.txt" 5 50
52    mediumWordsResults `seq` return ()
53    mediumEndTime <- getCPUTime
54    let mediumWordsTime = (mediumEndTime - mediumStartTime)
55
56    -- small word lengths test
57    smallStartTime <- getCPUTime
58    smallWordsResults <- simulateGames "data/medium_dict.txt" 3 50
59    smallWordsResults `seq` return ()
60    smallEndTime <- getCPUTime
61    let smallWordsTime = (smallEndTime - smallStartTime)
62
63    let testResult = smallWordsTime < mediumWordsTime && (mediumWordsTime <
          largeWordsTime)
64    assertBool "Smaller words should run faster than larger words!" testResult
65
66  testNumberGames :: Test
67  testNumberGames = TestCase $ do
68    -- many games test
69    manyStartTime <- getCPUTime
70    manyGamesResults <- simulateGames "data/medium_dict.txt" 6 75
71    manyGamesResults `seq` return ()
72    manyEndTime <- getCPUTime
73    let manyGamesTime = (manyEndTime - manyStartTime)
74
75    -- some games test
76    someStartTime <- getCPUTime
77    someGamesResults <- simulateGames "data/medium_dict.txt" 6 25
78    someGamesResults `seq` return ()
79    someEndTime <- getCPUTime
80    let someGamesTime = (someEndTime - someStartTime)
81
82    -- few games test
83    fewStartTime <- getCPUTime
84    fewGamesResults <- simulateGames "data/medium_dict.txt" 6 5
85    fewGamesResults `seq` return ()
86    fewEndTime <- getCPUTime
87    let fewGamesTime = (fewEndTime - fewStartTime)
88
89    let testResult = fewGamesTime < someGamesTime && (someGamesTime < manyGamesTime)
90    assertBool "Fewer game simulations should run faster than many simulated games!"
          testResult
```

8.8 Package.yaml

```
1  name:                hangman
2  version:             0.1.0.0
```

```
license:           BSD3
author:            Anthony Pitts (aep2195)

dependencies:
- base >= 4.7 && < 5
- random
- parallel
- split
- free
- containers
- HUnit
- deepseq

library:
  source-dirs: src
  ghc-options:
  - -Wall

executables:
  hangman:
    main:              Main.hs
    source-dirs:       app
    ghc-options:
    - -O2
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N2
    - -eventlog
    - -Wall
    dependencies:
    - hangman
    - random
    - parallel
    - split

tests:
  hangman-test:
    main:              RunTests.hs
    source-dirs:       tests
    ghc-options:
    - -O2
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N2
    - -eventlog
    - -Wall
    dependencies:
    - hangman
```