

Cheating Hangman in Haskell 98

Tom Moertel*

February 25, 2001

1 An introduction to “Cheating Hangman”

Rich Morin introduced me to “Cheating Hangman” on the *Fun With Perl* mailing list:

In the “hangman” game, the “dealer” presents the “player” with a set of character spaces to be filled in. In each move, the player offers a letter. If the letter matches any of the spaces in the dealer’s word, the dealer fills in those spaces with the letter. If the letter fails to match anything in the word, the dealer draws another element (e.g., head, neck) on a stick-figure representation of a “hanged” person. Eventually, either the player succeeds in determining the word, or an entire body has been drawn.

I don’t really know if it qualifies as cheating, but I have found that it splices up the “dealer” role quite a bit to leave the word initially undefined, save for the number of characters it contains. So, for example, I might draw six character positions and wait for the player to pick a letter. S/he chooses “e,” so I draw the head and eliminate all “e”-containing words from my candidate list.

As the game continues, my options get more and more limited; at some point, unless the player runs out of moves, I will find that I can no longer reject the player’s letter (i.e., all of the words that are left in my list have the proposed letter). I must then pick a set of positions and fill them in with the letter.

In general, I will have come down to a single word at this point, but this is not really a requirement. . . .

2 A Cheating Hangman game in Haskell

This variation on the classic Hangman game sounded fun indeed, but instead of implementing it in Perl, I used Haskell. What follows is the complete implementation, less than 100 lines of code, with unfolding documentation in *Literate Programming* fashion. I should note that I allow the player unlimited guesses—he can never be hanged—and provide a “verbose” mode that displays some of the program’s internal state as a visual aid to demonstrate the cheating logic.

*tom-haskell@moertel.com

2.1 Preliminaries

First we declare the module in which the program will live,

```
2a  <module declaration 2a>≡
      module Main (main) where
import a few modules for our use,
```

```
2b  <imports 2b>≡
      import List
      import Random
      import Char
      import System
      import GetOpt
```

and define a few types for convenience.

```
2c  <declarations 2c>≡
      type Words      = [ String ]
      type Guesses    = [ Char ]
      type GameState = ( Words    -- our "cheat list"
                        , Guesses  -- player's guesses
                        , Bool     -- whether we're being verbose
                        , StdGen ) -- random number generator
```

Defines:

```
  GameState, used in chunks 3-5.
  Guesses, used in chunk 7.
  Words, used in chunks 6 and 7.
```

2.2 The main program

The game is invoked like so:

```
cheating-hangman [-verbose] [-wordlen=wordlen] [-dict=dictionary]
where wordlen is the desired number of letters in the word to be guessed and dictionary is an optional path to a dictionary from which to obtain words for the game's "cheat list." The -verbose flag, if supplied, causes the game to display some of its internal workings.
```

The game handles its options via the `GetOpt` module. We define flags for each of the options the game handles and then describe how the command line is to be parsed.

```
2d  <options 2d>≡
      data Flag
          = OptVerbose | OptDict FilePath | OptLen String

      opts :: [ OptDescr Flag ]
      opts = [ Option ['v'] ["verbose"] (NoArg OptVerbose) "see game logic"
              , Option ['w'] ["wordlen"] (ReqArg OptLen "NUM") "use NUM-sized word"
              , Option ['d'] ["dict"]    (ReqArg OptDict "FILE") "use dict FILE" ]
```

Once invoked, the program gets the command line options, handles any usage errors that may have occurred, and then plays a single game by calling through to `hangman`.

```
3a  <main 3a>≡
    main :: IO ()
    main = do
      args <- getArgs
      pnam <- getProgName
      let header = "Usage: " ++ pnam ++ " [OPTION...]"
          case (getOpt Permute opts args) of
            (flags, [], []) ->
              let dictionary = case [d | OptDict d <- flags] of
                    [] -> "/usr/dict/words"
                    (d:_) -> d;
                  wordlen    = case [w | OptLen w <- flags] of
                    [] -> 6;
                    (w:_) -> (read w) :: Int
                  verbose    = not $ null [v | v@OptVerbose <- flags]
              in hangman wordlen dictionary verbose
            (_,_,errs) -> error (concat errs ++ usageInfo header opts)
```

Uses `hangman 3b`.

2.3 Playing a game

Next we prepare to play a game of Hangman. First, we obtain a new random number generator. Next, we pilfer all the words of the desired length from the supplied *dictionary* in order to form the word list from which we will play (or, to be perfectly honest, cheat). We then set up an initial game state, and finally start the game by playing the initial turn.

```
3b  <play a game 3b>≡
    hangman :: Int -> FilePath -> Bool -> IO ()
    hangman wordLen dictionary verbose = do
      rnd <- getStdGen
      dictWords <- readFile dictionary
      let gameWords = filter (\w -> length w == wordLen && all isLower w) $
              words $ dictWords
          state      = (gameWords, [], verbose, rnd) :: GameState
      playTurn state
```

Defines:

`hangman`, used in chunk 3a.

Uses `GameState 2c` and `playTurn 4a`.

2.4 Playing a turn

We start each turn by printing the current state of the game. Then, we check to see if the game is over. If it is, we end the game by printing out the number of guesses the player made. Otherwise, we ask for a guess, apply it to the state of the game (which yields a new state), and enter the next turn by passing the new state into a recursive call to `playTurn`.

```
4a  <play a turn 4a>≡
    playTurn :: GameState -> IO ()
    playTurn state@(ws, gs, _, _) = do
      putStrLn $ stateToStr state
      if gameOverQ state
        then do putStrLn $ "Game over in " ++ show (length gs) ++ " guesses."
        else do putStrLn "Your guess? "
              guess <- getLine
              case guess of
                [] -> do playTurn state -- player didn't guess -> redo
                g:_ -> if g `elem` gs
                      then do putStrLn $ "You already guessed '"+[g]++"'."
                            playTurn state
                      else do let (state', msg) = applyGuess state g
                                putStrLn msg
                                playTurn state'
```

Defines:

`playTurn`, used in chunk 3b.

Uses `applyGuess` 5b, `gameOverQ` 4b, `GameState` 2c, and `stateToStr` 5a.

2.4.1 Determining when the game is over

To determine if a game is over, we see if the first word in the cheat list is fully revealed by the player's guesses.¹ We can do this by filtering the characters of the word to remove any characters not in the list of guesses. If the resulting filtered word matches the original, the word has been fully revealed (i.e., no characters remain hidden), and the game is over.

```
4b  <determine if the game is over 4b>≡
    gameOverQ :: GameState -> Bool
    gameOverQ (ws, gs, _, _) =
      let firstWord = head ws in
      firstWord == filter ('elem' gs) firstWord
```

Defines:

`gameOverQ`, used in chunk 4a.

Uses `GameState` 2c.

¹I leave it as an exercise for the reader to show why checking only the first word in the list is sufficient.

2.4.2 Converting the game state into a string

In order to print out the state of the game, we convert it into a string. Note that for purposes of better exploring the game's inner workings, in verbose mode we include the entire word list in the string if it is short.

```
5a <convert the game state into a printable string 5a>≡
stateToStr :: GameState -> String
stateToStr (ws, gs, verbose, _) =
  let wordRep    = map (\c -> if c `elem` gs then c else '.') $ head ws
      wordsLeft  = show $ length ws
  in "\n" ++ wordRep ++ " [" ++ gs ++ "]"
    ++ if verbose
       then " (words=" ++ wordsLeft ++ "/" ++ (show $ cscore gs ws)
           ++ (if length ws <= 8 then " " ++ show ws else "") ++ ")"
       else ""
```

Defines:

`stateToStr`, used in chunk 4a.

Uses `cscore` 7 and `GameState` 2c.

2.5 Applying a guess to the game's state

When given a guess, the game will attempt to reject it by removing from its cheat list *ws* any words that would have been more fully revealed by the new guess. If the resulting list is empty, the game must accept the guess; otherwise the guess can be rejected.

But if the game can reject a guess, should it? Sometimes rejecting a guess will require the game to reduce the cheat list to the extent that future cheating is severely hampered. A better approach, and what we use below, is to try rejecting *and* accepting the player's guess, settling on whichever is better. Which is better is determined by a scoring function `cscore`, described later.

```
5b <apply a guess g to the game's state 5b>≡
applyGuess :: GameState -> Char -> (GameState, String)
applyGuess (ws, gs, verbose, rnd) g =
  let gs'      = sort (g:gs)
      score    = cscore gs'
      (ws', nws') = partition (g `notElem` ws) -- try rejecting g
      (ws'', rnd') = findBestSublist nws' g rnd score -- try accepting g
      (bestws, msg) = if score ws' > score ws'' -- which is best?
                      then (ws', "Sorry!") -- rejecting is best
                      else (ws'', "Good guess!") -- accepting is best
  in ((bestws, gs', verbose, rnd') :: GameState, msg)
```

Defines:

`applyGuess`, used in chunk 4a.

Uses `cscore` 7, `findBestSublist` 6, and `GameState` 2c.

2.6 Keeping the best part of our cheat list

If the player guesses the letter g , and all the words in our cheat list contain it, we have no choice but to accept the guess. What is worse, we must reveal to the player *where* in our “hidden word” g is positioned. Most likely, some of the words in our list will not have g in the positions we choose to reveal, and those words must be removed, reducing our opportunity to cheat later.

For example, if our cheat list were ["forces", "stones", "stumps"] and the player's guess were 's', we might choose to reveal that the guessed letter occupies the 5th position (counting from zero, starting from the left) of our hidden word: ". . . . s". Of our list's words, only "forces" matches this positional signature, and so the other two would have to be removed from our cheat list. We would be better off revealing positions 0 and 5 ("s . . . s") because then we could keep two words in our cheat list—["stones", "stumps"].

The function below refines this thinking by using a scoring function (provided by the caller) to rate how much “cheating opportunity” a word list provides. First, it divides the overall cheat list into groups having identical positional signatures, based on where the guessed letter occurs in each word,

```
[ [(0,5), "stones"], (0,5), "stumps"]
, [(5), "forces"] ]
```

then it scores each group,

```
[ (8, ["stones", "stumps"])
, (5, ["forces"]) ]
```

then selects the most highly scored groups (ties are possible, but there is only one best group in this example),

```
[ ["stones", "stumps"] ]
```

and finally chooses one of the groups at random (to break ties).

```
["stones", "stumps"]
```

```
6 <find the best subset of our word list 6>≡
  findBestSublist :: Words -> Char -> StdGen -> (Words->Int) -> (Words, StdGen)
  findBestSublist [] _ rnd _ = ([], rnd) -- handle trival case
  findBestSublist ws g rnd score =
    let wsGroups      = groupfst . sortfst . map (\w->(elemIndices g w, w)) $ ws
        scoredGroups = map ((\wsg -> (score wsg, wsg)) . map snd) wsGroups
        maxScore     = maximum $ map fst scoredGroups
        bestGroups   = map snd . filter ((==maxScore).fst) $ scoredGroups
        (pick, rnd') = randomR (0, length bestGroups - 1) rnd
        bestSublist  = bestGroups !! pick
    in (bestSublist, rnd')
  where groupfst = groupBy (\a b -> (fst a) == (fst b))
        sortfst  = sortBy (\a b -> compare (fst a) (fst b))
```

Defines:

findBestSublist, used in chunk 5b.

Uses Words 2c.

2.7 Scoring a cheat list

Given a cheat list *ws* and guesses *gs*, we can compute a score that describes how much opportunity for future cheating the list provides. The scoring heuristic we use is simply to count the unique letters in each of *ws*'s words, skipping those letters that have been guessed, and adding the counts together to yield the overall score. This heuristic favors longer, more varied lists to shorter ones whose words contain many repeated characters. For example, assuming that no guesses have been made, ["scores", "stones"] has a score of 10, and ["frolic", "stones"] scores 11.

```
7 <determine the "cheat score" of a word list ws, given guesses gs >≡
  cscore :: Guesses -> Words -> Int
  cscore gs = sum . map (length.group.sort.filter('notElem'gs))
Defines:
  cscore, used in chunk 5.
Uses Guesses 2c and Words 2c.
```

3 Index of key definitions and functions

applyGuess: 4a, [5b](#)
 cscore: 5a, 5b, [7](#)
 findBestSublist: 5b, [6](#)
 gameOverQ: 4a, [4b](#)
 GameState: [2c](#), 3b, 4a, 4b, 5a, 5b
 Guesses: [2c](#), 7
 hangman: 3a, [3b](#)
 playTurn: 3b, [4a](#)
 stateToStr: 4a, [5a](#)
 Words: [2c](#), 6, 7

4 A sample game of Cheating Hangman

Here's what a sample game looks like. When the game prints “words= c/s ”, c gives the count of words in the cheat list, and s gives the list's current score.

```
$ ./cheating-hangman --wordlen=6 --verbose
```

```
..... [] (words=4962/26783)
```

```
Your guess? a
```

```
Sorry!
```

```
..... [a] (words=2901/15456)
```

```
Your guess? e
```

```
Sorry!
```

```
..... [ae] (words=830/4456)
```

```
Your guess? i
```

```
Sorry!
```

```
..... [aei] (words=374/1971)
```

```
Your guess? o
```

```
Sorry!
```

```
..... [aeio] (words=101/532)
```

```
Your guess? y
```

```
Sorry!
```

```
..... [aeioy] (words=57/292)
```

```
Your guess? u
```

```
Good guess!
```

```
..u... [aeiouy] (words=36/152)
```

```
Your guess? l
```

```
Sorry!
```

```
..u... [aeilouy] (words=24/100)
```

```
Your guess? r
```

```
Good guess!
```

```
.ru... [aeiloruy] (words=12/42)
```

```
Your guess? n
```

```
Sorry!
```

```
.ru... [aeilnoruy] (words=7/23)
```

```
Your guess? s
```

```
Good guess!
```

.ru..s [aeilnorsuy] (words=4/11 ["crumbs","trucks","trumps","truths"])

Your guess? *h*

Sorry!

.ru..s [aehilnorsuy] (words=3/9 ["crumbs","trucks","trumps"])

Your guess? *b*

Sorry!

.ru..s [abehilnorsuy] (words=2/6 ["trucks","trumps"])

Your guess? *c*

Sorry!

.ru..s [abcehilnorsuy] (words=1/3 ["trumps"])

Your guess? *p*

Good guess!

.ru.ps [abcehilnoprasy] (words=1/2 ["trumps"])

Your guess? *m*

Good guess!

.rumps [abcehilmnoprasy] (words=1/1 ["trumps"])

Your guess? *t*

Good guess!

trumps [abcehilmnoprstuy] (words=1/0 ["trumps"])

Game over in 16 guesses.