Deciphering Haskell monadic `do`

John Lusk

February 10, 2019

Abstract

A stupid litle exercise in which I try to figure out how `do` really works in the solution presented in the "knight's quest" section of the chapter "A Fistful of Monads" of the book *Learn You a Haskell For Great Good!*.

The book treats lists as "indeterminate answers" (i.e., a list of possible values that the "answer" could be, as opposed to a single "answer" value).

It also uses `List` as a `Monad`, along with `do`, so I wanted to try to figure all this out.

1 Original Code, with Some Re-writing

I start with the code from *Learn You a Haskell*, and do a little re-writing to eliminate `do`.

1.1 Main Code

Some initial setup:

```
module KnightMoves where

import Control.Monad

-- Modules for unit test
import Test.Hspec
import Data.Set as Set

-- | (Column, Row)
type KnightPos = (Int,Int)
```

Here is the original definition of `moveKnight` from the book:

```
-- | The original function from Learn You a Haskell
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = do
    (c',r') <- [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
               ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
               ]
    guard (c' `elem` [1..8] && r' `elem` [1..8])
    return (c',r')
```

What the heck is is going on here? The first thing I did was to move that long list out of the code body, using a `where` clause to simplify things, hence the `w` in the function name:

```
-- | "moveKnight where" -- first re-write
mkw :: KnightPos -> [KnightPos]
mkw (c,r) = do
  (c',r') <- moves
  guard (c' `elem` [1..8] && r' `elem` [1..8])
  return (c',r')
  where
    moves = [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
               ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
               ]
```

Then, we rewrite to translate the `do` into what it represents (`>>=` and `>>` operators).

We also define a special `gard` function that does the same thing as `guard`. Remember, `guard`'s job is to yield either `mzero` (from `MonadPlus`), which is just `[]` in this case, or `return ()`, which, in the case of the `List` monad, is just `[()]`, the list containing a single empty tuple (also known as "unit").

The use of `>>` after `gard` will either substitute another value for the incoming value (if it's `[()]`, or fail to do anything if the incoming value is the empty list `[]` (because we're mapping a function to the incoming list, which, if empty, results in an empty list, no matter what the function is).

Here's the code:

```
-- | 2nd rewrite, without "do"
mkw2 :: KnightPos -> [KnightPos]
mkw2 (c,r) =
  -- do e1 ; e2      =        e1 >> e2
  -- do p <- e1; e2  =        e1 >>= \p -> e2
  moves >>= \(c',r') -> gard (c',r') >> [(c',r')]
  where
    moves = [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
               ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
               ]
    gard (c',r') = if (c' `elem` [1..8] && r' `elem` [1..8])
                   then [()]
                   else []
```

Then we do the exact same thing with another function, `in3`:

```
in3 :: KnightPos -> [KnightPos]
in3 start = do
    first <- mkw2 start          -- was: moveKnight, not mkw
    second <- mkw2 first
    mkw2 second
```

The rewrite of this function looks pretty much the same as that for `moveKnight`, except there is no occurrence of the `>>` operator:

```
-- | Re-write, without "do"
in32 :: KnightPos -> [KnightPos]
in32 start =
  mkw2 start >>= \first -> mkw2 first >>= \second -> mkw2 second

canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

1.2 Unit Test

The best way, in the long run, to test that we've rewritten the functions properly, as we iteratively edit and test our code, is to simply assert that we have done so. To that end, we write easily-repeatable test code.

```
main :: IO ()
main = hspec $ do
  describe "various implementations" $ do
    it "moveKnight == mkw" $
```

(Here follows the actual assertion:)

```
      (Set.fromList $ moveKnight (5,5)) == (Set.fromList $ mkw (5,5))
    it "mkw == mkw2" $
      (Set.fromList $ mkw (5,5)) == (Set.fromList $ mkw2 (5,5))
    it "in3 == in32" $
      (Set.fromList $ in3 (5,5)) == (Set.fromList $ in32 (5,5))
```

2 Explanation of How It All Works

2.1 `MoveKnight` (or `mkw2`)

As you recall, the definitions for `>>=` and `>>` (for `List`) are as follows:

```
xs >>= f = concat (map f xs)}
xs >> ys = xs >>= \_ -> ys
```

`>>` can be rewritten as `xs >> ys = concat (map (_ -> ys) xs)`. You can see that `>>` will substitute `ys` for `xs` only if `xs` is non-empty (or non-`fail`, since `fail` is the empty list). Otherwise, `>>` does nothing, and the result is still `[]`.

In `moveKnight`, we feed an entire list of (possible) knight positions (column, row) into a function, which, *for each element of the list* (by virtue of `map`), uses the `guard` (`gard`) function to yield either `[]` (if the condition fails) or `[()]`, which is *not* an empty list (if the condition succeeds). That `guard` result is then fed through `>>`, which attempts to substitute a single-element list containing the original input value (`(c',r')`, an element of the original list of knight positions). That substitution will fail if `guard` failed, since mapping a function to an empty list results in an empty list, no matter what the function is.

The result of this giant function mapped to a list of knight positions is a list of singleton lists that looks something like this:

```
[ [(1,1)], [(2,2)], [], [(4,4)] ]
```

We "flatten" this list out by applying `concat` to it, which concatenates all the lists into one big list, resulting in

```
[ (1,1), (2,2), (4,4) ]
```

2.2 `in3` or `in32`

`in3` (`in32`) begins with the result of `moveKnight` (`mkw2`), a monad (`List`), and binds it (`>>=`, "bind") to a giant function. This function is applied to *each member* (labelled `first` of the list generated by `moveKnight`, per the definition of `>>=`.

`first` is itself passed to `moveKnight` (again), which generates a *new* list of positions which is in turn bound to a new (nested) function, which processes each element of the moves generated from `first`. (The elements of the second list are named `second` when passed to the bound function.)

`moveKnight` is called for a third time on the elements of the second list (each elementwise named `second`), to generate a possible list of moves from the second move.

At the end of all this, we've called `moveKnight` three times, constantly expanding the list of possible 3-move positions of the knight. (Remember, we're calling `concat . map` all the time, so we're constantly flattening lists of lists.)

At the end of all this, we have a list of moves possible by moving the knight three times.