

Memoization in Haskell

Henry Laxen

April 3, 2008

I have struggled many a time with memoizing something or other in Haskell, and I have to admit that the brief section found on the [haskell.org](http://haskell.org/wiki) wiki site was usefull, but certainly not transparent. Here is their *easy* way to memoize the fibonacci sequence

The Wiki Version 1

Haskell Code

```
1
2 > slow_fib :: Int -> Integer
3 > slow_fib 0 = 0
4 > slow_fib 1 = 1
5 > slow_fib n = slow_fib (n-2) + slow_fib (n-1)
6 >
7 > memoized_fib :: Int -> Integer
8 > memoized_fib =
9 >   let fib 0 = 0
10 >       fib 1 = 1
11 >       fib n = memoized_fib (n-2) + memoized_fib (n-1)
12 >   in (map fib [0 ..] !!)
```

This works great, and it is very simple. The idea is to look up the previous value of the fibonacci function in a list, rather than recompute it. The problem is that you have to know in advance what the arguments are that the fibonacci function will be called with. In the case of fibonacci that is pretty easy, but what if you don't know the arguments in advance? This simple technique isn't going to work.

The wiki goes on to mention something about fixed points and recursion, which I have to admit I didn't understand.

The Wiki Version 2

Haskell Code

```

14
15 > fib :: (Int -> Integer) -> Int -> Integer
16 > fib f 0 = 1
17 > fib f 1 = 1
18 > fib f n = f (n-1) + f (n-2)
19 >
20 > fibonacci :: Int -> Integer
21 > fibonacci = memoFix fib
22 >
23 > memoFix :: ((a -> b) -> (a -> b)) -> a -> b
24 > memoFix f =
25 >   let mf = memoize (f mf) in mf

```

I suppose the `memoize` function is left as an exercise for the reader, but this reader didn't quite get it. Also the role of `f` as an argument in `fib` didn't make sense to me.

It wasn't until I found this post by Dick Thierbach discussing the memoization of the Ackerman function that things finally began to make sense. I hope the explanation that follows will spare you, gentle reader, the hours of frustration I endured to finally get to the bottom of this memoization thing in Haskell.

Memoize in All Generality

Haskell Code

```

27
28 > import Debug.Trace
29 > import Data.Map as Map
30 > import Control.Monad.State.Lazy as State
31 >
32 >
33 > tfib m n = trace ("fibM called with " ++ show m ++ " returning " ++ show n) n
34 >
35 > fibM :: Monad m => (Integer -> m Integer) -> Integer -> m Integer
36 > fibM f' 0 = return $ tfib 0 1
37 > fibM f' 1 = return $ tfib 1 1
38 > fibM f' n = do
39 >   a <- f' (n-1)
40 >   b <- f' (n-2)
41 >   return $ tfib n (a+b)
42 >
43 > type StateMap a b = State (Map a b) b
44 >
45 > memoizeM :: (Show a, Show b, Ord a) =>
46 >   ((a -> StateMap a b) -> (a -> StateMap a b)) -> (a -> b)
47 > memoizeM t x = evalState (f x) Map.empty where
48 >   g x = do
49 >     y <- t f x
50 >     m <- get
51 >     put $ Map.insert x y m
52 >     newM <- get
53 >     return $ trace ("Map now contains\n" ++ Map.showTree newM) y
54 >   f x = get >>= \m -> maybe (g x) return (Map.lookup x m)
55 >
56 >
57 > fib n = memoizeM fibM n
58 >

```

Let's look at this, a little at a time. The interesting function is `memoizeM`, so we'll start there. `evalState` lives in the `Control.Monad.State.Lazy` library,

and make it easy to keep an internal state within its “sphere.” The internal state we are keeping, in this instance, is a map of all of the previous calls to the `fibM` function, as well as the results it returns. Let’s take a closer look at how this works. We start off with: `evalState (f x) Map.empty` where `f` is defined as: `f x = get >>= \m -> maybe (g x) return (Map.lookup x m)`. `f` gets the internal state, which is the current map, (initially empty of course.) It then tries to lookup `x` in this map. If `Map.lookup` succeeds, returning a `Just n` value, we are done, and strip the `Just` off with the `maybe` function. Life is good. If `Map.lookup` returns `Nothing`, the `maybe` function returns is *default value*, which in this case is `(g x)`. So next we must explore what `g` is doing.

`g` is actually very simple, though there is one subtle feature that I was too dense to appreciate without some mental gymnastics. The subtlety appears right off the bat in line 46, namely `y <- t f x`. `t` is the function that we were originally called with, in this case it is `fibM`. `t` is called with the function we just looked at, namely `f` as its first argument, and `x`, the value we are looking for, ie. the *x*th Fibonacci number, as the second argument. So, lets look at `fibM` and bathe in the glory of enlightenment. The first argument to `fibM` is a function, which in this case is our old friend `f`. But `f` role in life is to lookup a previously computed result, and return it. Since `f` is being called with `(n-1)` and `(n-2)`, this result will exist and be returned immediately. So now we finally understand why `fibM`’s first argument is a function, so we can *thread* the memoization function through it, grab the arguments and the result, and **memoize it!**. The rest of `g` is just gravy now. In line 47 we retrieve our map, an in line 48 we insert the argument and the result into the map. The beauty of this approach is that we do not have to know the arguments in advance, as we did in the first version of the memoized fibonacci function. Furthermore, this approach will work with any function we want to memoize. The scheme is to rewrite the function so that it takes exactly two arguments. The first is a function which will do the memoization, and the second will be a tuple of all of the required arguments for the function. That way, the tuple will be the key in the map, and whatever the function returns will be the value. We **do not need to know the arguments in advance**. The `memoizeM` function takes care of keeping track of those for us.

And the Answer Is

Here is the output of the above using the wonderful `Debug.Trace` module to allow us to watch the whole thing in action. Notice that the map is built up piece by piece, as the fibonacci function is called, not all at once for all possible arguments. This may not seem like a big deal for fibonacci, but for Ackerman, arguments get big, very big, and using a list or an array to hold the lookup table just isn’t feasible. Life is beautiful, once again.

```
*Main fib 5
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
fibM called with 1 returning 1
fibM called with 0 returning 1
Map now contains
1:=1

Map now contains
1:=1
+—0:=1
+—|

fibM called with 2 returning 2
Map now contains
1:=1
+—0:=1
+—2:=2

fibM called with 3 returning 3
Map now contains
1:=1
+—0:=1
+—2:=2
+—|
+—3:=3 p

fibM called with 4 returning 5
Map now contains
1:=1
+—0:=1
+—3:=3
+—2:=2
+—4:=5

fibM called with 5 returning 8
Map now contains
1:=1
+—0:=1
+—3:=3
+—2:=2
+—4:=5
+—|
+—5:=8

8
Main
```

Conclusion

Well, gentle reader, there you have it. At this point I hope it is obvious how to memoize just about any function you can dream up with Haskell. If there is something in this article that you feel needs further explanation, please drop me a line at *nadine.and.henry@pobox.com* and I'll do my best to clarify things. I will leave you with a thought that should be emblazoned deep within the brains of just about every programmer I know. **Better is the enemy of good.** The rest is silence.

This file is also available in pdf form as well as plain text (markdown) for your reading pleasure



Henry Laxen