Assignment N2: Memoization

Noric Couderc

Introduction

In this assignment, we'll look at how recursive functions can be slow, and how we can make them faster.

We will look at two problems:

1. Calculating the nth element of the fibonacci sequence, which is usually used as a simple example to show the problem with recursion
2. Calculating the longest palindromic subsequence of a string, which is the longest sequence of letters taken from the input string where
    1. The sequence is a palindrome (if you reverse it, you get the same sequence)
    2. If two letters are in a certain order in the result, they are in the same order in the input. So if the sequence is "aba", you should have an "a" before the "b" in the input, and another "a" after the "b". With possibly some letters in between.

Here are some examples of the longest palindromic subsequence:

- **kayak** -> kayak
- w**ri**te**r**s -> rir
- af**e**rociou**sm**on**ada**te**m**yham**s**t**e**r -> esmadamse

Provided Files

- `app` : Directory for the main program
    - `Main.hs`: The main program
- `src`: Memoization library
    - `Memoization.hs`: Main file to modify
- `test`: Tests
    - `Spec.hs`: Testing with QuickCheck
- `bench`: Benchmarks
    - `Bench.hs`: Benchmarking with Criterion
- `README.md`: The instructions
- `README.pdf`: The instructions (in PDF)
- Generated files:

- package.yaml: Dependencies, etc.
- cabal.project.local
- CHANGELOG.md
- LICENSE
- memoization.cabal
- Setup.hs
- stack.yaml
- stack.yaml.lock

What to Submit

Submit the file `Memoization.hs`. To pass the assignment, your code must:

1. Work correctly: use `stack test` to check your code passes our tests.
2. Be fast enough: use `stack bench` to benchmark the different functions.

How to Submit

Use the course Canvas for that purpose

Deadline

- You should submit your file on the 12/5/2025 at the latest.

Fibonacci

The famous fibonacci function is a typical example of a recursive function.

```haskell
fibo :: Int -> Int
fibo 0 = 0
fibo 1 = 1
fibo n = fibo (n-1) + fibo (n-2)
```

Define the fibonacci function in the file `src/Memoization.hs`.

You can load the module by running `stack ghci`, it will give you access to the interpreter (`ghci`).

In `ghci`, type:

```
> :set +s
> fibo 35
```

It will run the function and print how long it takes. You'll see the function is quite slow.

The problem with this function is that it re-calculates values every time it's called, and since it calls itself, that makes a lot of calls. To see what the problem is, you can use the `trace` method in the `Debug.Trace` module, it's basically the good old `print` you're used to. If you trace `n` before computing the result, and call `fibo 7`, you will see the same numbers appearing many times, every time the number appears, we have to recalculate everything.

To avoid this, we will make a *cache*. Caching is a simple idea: Instead of recalculating, we will store the results in a data-structure, and just look them up when we need them. That way, if we need the value of the function, we can look in the cache instead.

Look at the file `src/Memoization.hs` try to understand how the function `fastFibo1` does. You can see how the cache is growing by writing

```
-- Prints just 'fiboCache = (_t1::[Integer])'
-- since the list isn't evaluated yet
> :print fiboCache
> fastFibo1 10
> :print fiboCache
```

Now, we will make a more general version of this optimization, so that it works on other types (e.g. strings), instead of integers. So it will not use indexing to look into the cache, but instead work with a list of key-value pairs.

Write two functions, `listCache` and `listLookup`.

1. `listCache` builds a cache as a list of tuples: It takes a domain and a function `f`, and creates a list of tuples `(x, y)`, where `x` is from the domain, and `y = f x`.
2. `listLookup` takes a list of tuples (a cache) and an `x`, it returns the value matching that `x` in the tuples.

```haskell
listCache :: [a] -> (a -> b) -> [(a, b)]
-- Example
-- listCache [1, 2, 3] (\x -> 2*x) ==> [(1,2), (2,4), (3,6)]

listLookup :: Eq a => [(a, b)] -> a -> b
-- Example
-- listLookup [('a', 'b'), ('c', 'd')] 'c' ==> 'd'
```

Usually, the `listLookup` function would return a `Maybe b`, but we can ignore that for now, you can use `fromJust` to go around this problem.

The Ouroboros

Now, we will do something a bit surprising. We'll write two functions, that call each other.

1. **Write** a function `fibCache` which builds an (infinite) cache using `listCache` and the `fibo` function.
2. **Write** a function `fastFibo2 :: Int -> Int` which takes and `x` and uses `listLookup` to look up the value for `x` in `fibCache`.

3. **Test** `fastFibo2`, is it much faster than `fibo`? Write down how long it takes in a comment.
4. Now, **change** the `fibCache` function, so that it uses `fastFibo2` instead of `fibo`.

If you call `fastFibo2` now, it should be fast.

The idea is that we have two functions that call each other:

1. `fibCache` builds a cache, using `fastFibo2`, since it has the right type, it works.
2. `fastFibo2` doesn't compute much, it just looks in the cache!

The secret lies in two things:

1. The cache we build is not *really* infinite, Haskell will build it as needed. You can actually see the current state of the cache by typing `:print fibCache` in GHCI.
2. The `fastFibo2` function refers to `fibCache`, and Haskell will then uses the same exact pointer for the cache, we say the cache is *shared* between calls.

If this is difficult, one thing that helps is to make a finite cache first (e.g. 100 integers), and check if it works. And then make the cache infinite later.

Tightening the Knot

Now, look at the types of `listCache` and `listLookup`:

```haskell
listCache :: [a] -> (a -> b) -> [(a, b)]
listLookup :: Eq a => [(a, b)] -> a -> b
```

We see that they're pretty much the inverse of each other: One takes a function and makes a list, the other takes a list and makes it a function.

We write a function:

```
memoizeList :: Eq a => [a] -> (a -> b) -> (a -> b)
memoizeList domain = listLookup . (listCache domain)
```

Which makes a cache and looks immediately in it.

If we try to use this function to accelerate `fibo` by writing `memoizeList [0..] fibo`, it will still not work. This is because it would wrap *around* the fibo function, but the fibo function still calls the slow version.

To fix this, we will use **open recursion**.

Open Recursion

An open recursion is when we take a recursive function and make it "optionally recursive", so to speak. Instead of calling itself, we pass a function as an argument and call that function instead.

Write the function:

```
openFibo 0 = 0
openFibo 1 = 1
openFibo n = openFibo (n-1) + openFibo (n-2)
```

This is just the Fibonacci function again, but we'll try to remove the recursion. How? We just need to make it call *another* function, which we call `f` which we pass as an argument.

```
openFibo _ 0 = 0
openFibo _ 1 = 1
openFibo f n = f (n-1) + f (n-2)
```

Now, `openFibo` takes a function `f` and an integer `n`, and instead of calling itself, it calls `f`. Now, this function could do *more* than the original, is not necessarily recursive anymore, and we *can* get an equivalent to the Fibonacci function.

```
-- Calling openFibo with itself as an argument
-- (Probably not intuitive, but it works!)
-- And we get the recursive version.
fibo :: Int -> Int
fibo = openFibo fibo

fibo n -- Use: definition of fibo
= openFibo fibo n -- Use: definition of openFibo
= fibo (n-1) + fibo (n-2)
```

Now, we can make the fast fibonacci again, by writing:

```
fastFibo3 = memoizeList [0..] (openFib fastFibo3)

-- And that is because
openFibo fastFibo3 0 = 0 -- No recursion
openFibo fastFibo3 1 = 1 -- No recursion
-- Calls the fast versions!
openFibo fastFibo3 n = fastFibo3 (n-1) + fastFibo3 (n-2)
```

Test the fibonacci function, it should now be fast enough.

Longest Palindromic Subsequence

We'll apply the same technique to another function, which calculates the "longest palindromic subsequence". It takes a string as an input, and calculates the longest sequence of characters in that string which forms a palindrome.

Again, the examples:

- w**r**it**er**s -> rir
- **kayak** -> kayak

- af**e**rocious**sm**on**ada**te**m**yham**s**t**e**r -> esmadamse

Write some other examples of what the function should do.

Write a function `lps` which calculates this.

1. An empty string is already a palindrome.
2. A string with one letter is also a palindrome.
3. If the string is longer, we look at the first and last letters.
   1. If they are the same, we look for the LPS of the middle, and add the two characters on the sides.
   2. Otherwise, we can either drop the first character, or the last, try to find the LPS of each, and keep the longest of the two.

(Of course, in case 3.1 and 3.2, the function should be recursive).

This function takes strings and returns a string. So we need a data structure to store lists, which works as a cache. We will use a Trie (or prefix tree).

Tries

First, we need a definition for a type Trie.

```
data Trie node edge = Trie node [(edge, Trie node edge)]
  deriving Show
```

We will need to build a trie which can store *all* possible strings. But before we'll build that, we need to get used to the type, as it is a bit complicated.

Here are some examples of tries:

```
verySmall = Trie 0 [] -- Trie with value 0, no children

-- Trie that looks like
--       0
--   - /   \ +
--    -1    +1
-- (Two children with values -1, and +1, and two labels on the edges)
-- The children have no children themselves
small = Trie 0 [('-', Trie (-1) []), ('+', Trie (+1) [])]

-- Trie that contains the words "hi" and "ho" with the length of the strings
hiho = Trie 0 [
   ('h', Trie 1 [('i', Trie 2 []),
                 ('o', Trie 2 [])])
   ]
```

Looking Up In The Cache

This time, we will start with the function which looks up in a Trie.

The signature of the function is as follows:

```
trieLookup :: Eq e => Trie a e -> -- the trie
   [e] -- the list of edges to follow
   -> a -- The value of the node we end up at.
```

There are two cases:

1. If the list is empty, we return the value of the current node
2. If the list is not empty, we find the subtree to explore using the current character, and do a lookup from there.

Write the function `trieLookup`, come up with examples of its usage if you are not sure.

Mapping Over A Trie

We will also need to map a function to the cache (to make it a cache of our function).

Write the function `mapTrie` which maps a function over all the nodes of the Trie.

```
mapTrie :: (a -> b) -> Trie a e -> Trie b e
```

Building An Infinite Cache

Now, we need to build an infinite cache of strings. In this part, we will build Haskell code which would not work at all in other mainstream programming languages, so it might be confusing at first.

We will build it with three functions:

1. `rootTrie` takes a domain and builds the root of the trie, which has an empty list as a value and a list of edges, built with `edges`
2. `edges` builds the list of edges, where each edge has:
    1. A label, drawn from the domain
    2. A subtree, which is a node starting at this position in the tree.
3. `subtree` builds a subtree, it needs three values, the domain, the edge that was just followed, and the value of the parent node. It should call `edges`.

The tree is infinite because `edges` calls `subtree`, and `subtree` calls `edges`.

To debug your code, you may use the function `limitTrie` to make a Trie of finite length, that makes it easier to inspect, for example:

```
ghci> limitTrie 1 $ rootTrie ['a', 'b', 'c']
Trie "" [('a',Trie "a" []),('b',Trie "b" []),('c',Trie "c" [])]

ghci> limitTrie 2 $ rootTrie ['a'..'b']
Trie []
```

```
[('a',Trie "a" [('a',Trie "aa" []),('b',Trie "ab" [])]),
 ('b',Trie "b" [('a',Trie "ba" []),('b',Trie "bb" [])])]
```

You can also use `:print` in GHCI, for pretty printing.

Finishing Up

Then, the last steps are:

1. **Write** the function `trieCache`, which builds a cache for a function, using `mapTrie` and `rootTrie`.
2. **Write** the `openLPS` function, which uses open recursion
3. **Write** the `fastLPS` function, which uses `openLPS` and `trieCache`.
4. *Optional*: Write a fast version of fibonacci or fastLPS, using Maps and the State Monad.

Testing it works

To check if our implementation works, we need to test two things:

1. Does the optimization return the same results as the (slow) reference implementation?
2. Does the optimized version really run faster than the original?

Testing For Correctness

The file `test/Spec.hs` contains the tests, and is heavily commented.

It uses a library called QuickCheck: The user defines *properties* that the functions should satisfy, and QuickCheck generates random data to test if the properties hold. If QuickCheck finds a counter-example, it tries to *shrink* it (find a smaller counter-example) and reports it to the user.

You may look at the official documentation for more information. QuickCheck's official manual is a bit outdated, but still useful.

Look at the file `test/Spec.hs`, try to understand what it does.

You can run the file using `stack test`.

Testing Performance

For testing performance, we'll use a library called Criterion. The file which specifies the benchmarks is in `bench/Bench.hs`. It is also commented.

To run the benchmarks, run:

```
stack bench --benchmark-arguments="--output bench.html"
```

It will run the benchmarks (takes about 30 seconds on my machine) and will produce and HTML report, with interactive plots. This page explains what the plots mean.