Damian Franco

dfranco24@unm.edu

101789677

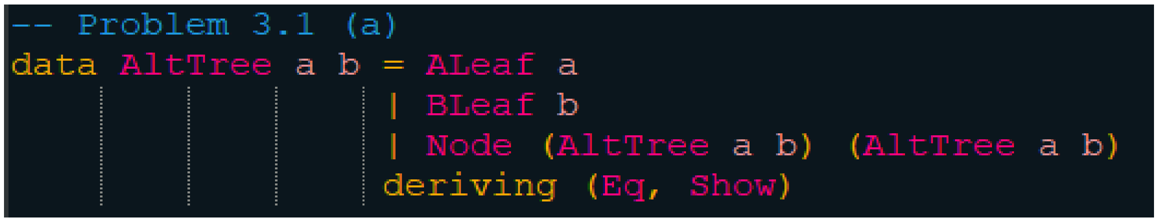
CS-558 001

Homework 3

3.1 Polymorphic tree datatypes

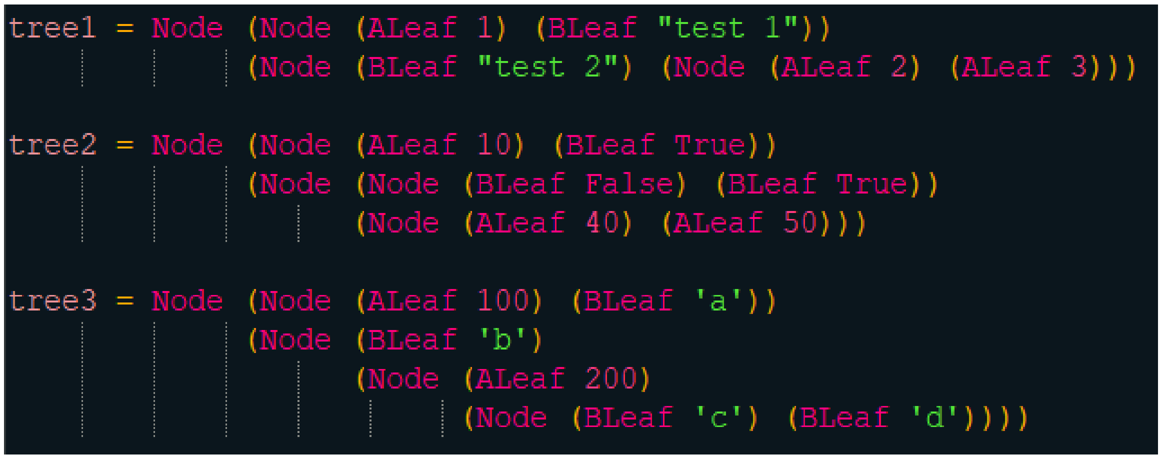
a) For the first question, we were asked to create a new data type for a tree within Haskell. This tree implementation is meant to be a binary tree that holds two separate kinds of leaves. Leaves will be annotated and hold a value. Each leaf would hold two separate types, for example some leaves in a tree can hold type *Int* and the rest of the leaves will hold the type *[Char]*. This can allow for more involved arithmetic between two separate types within a single tree. Each node within the tree should *not* be annotated which means nodes will not hold any values.

The way that I went about implementing this was by referencing the single-type binary tree examples we went over in class and throughout various resources on the internet. I first took the implementation that Dr. Lakin showed us in class a tree and added another leaf type to represent the two different types of leaves that this tree can have. *ALeaf* and *BLeaf* represent the types *a* and *b*. Since each *Node* did not hold a value, then the handling of that was to make this type recursive on itself by passing the node branches as arguments and stopping when a leaf occurs. Below you can view the code I used to implement this tree.

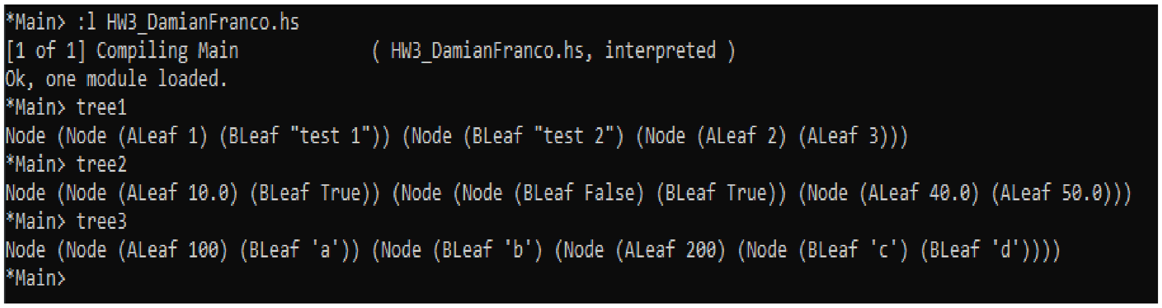


```haskell
-- Problem 3.1 (a)
data AltTree a b = ALeaf a
                 | BLeaf b
                 | Node (AltTree a b) (AltTree a b)
                 deriving (Eq, Show)
```

Testing this tree implementation was fairly simple. I created three trees with various different types. *Tree1* is a tree with type *a* as a *Int* and type *b* of the tree is a *[Char]* or *String*. *Tree2* has types *Double* and *Boolean*, and *Tree3* has types *Int* and *Char*. This allowed me to check if most types were compatible with the write up of the tree. Of course to fully test if the data type was implemented correctly, it will take printing out the tree to the console which is why I also derived *Show* to "pretty print" into the console. The data type also derives *Eq* which will help in later parts of this homework when testing for equality between leaves. These three trees will be used to test all functions that are written in the next parts of the homework as well. Below you can see the code I wrote to make the three trees and how they print to console.



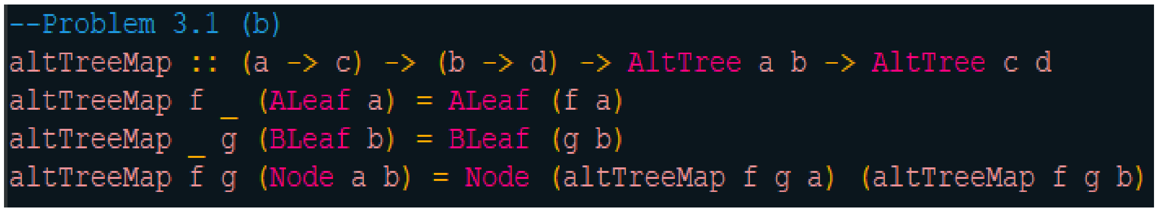
```haskell
tree1 = Node (Node (ALeaf 1) (BLeaf "test 1"))
             (Node (BLeaf "test 2") (Node (ALeaf 2) (ALeaf 3)))

tree2 = Node (Node (ALeaf 10) (BLeaf True))
             (Node (Node (BLeaf False) (BLeaf True))
                   (Node (ALeaf 40) (ALeaf 50)))

tree3 = Node (Node (ALeaf 100) (BLeaf 'a'))
             (Node (BLeaf 'b')
                   (Node (ALeaf 200)
                         (Node (BLeaf 'c') (BLeaf 'd'))))
```



```
*Main> :l HW3_DamianFranco.hs
[1 of 1] Compiling Main             ( HW3_DamianFranco.hs, interpreted )
Ok, one module loaded.
*Main> tree1
Node (Node (ALeaf 1) (BLeaf "test 1")) (Node (BLeaf "test 2") (Node (ALeaf 2) (ALeaf 3)))
*Main> tree2
Node (Node (ALeaf 10.0) (BLeaf True)) (Node (Node (BLeaf False) (BLeaf True)) (Node (ALeaf 40.0) (ALeaf 50.0)))
*Main> tree3
Node (Node (ALeaf 100) (BLeaf 'a')) (Node (BLeaf 'b') (Node (ALeaf 200) (Node (BLeaf 'c') (BLeaf 'd'))))
*Main>
```

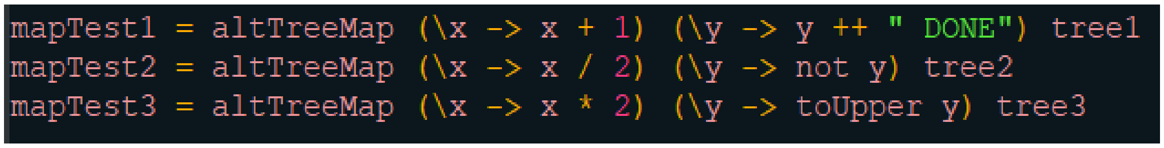
b) The next part of this first problem asked us to implement a polymorphic map function that will perform two mapping functions *f* and *g* over the tree. The mapping function *f* would be applied to any tree that is of type *a*, so every *ALeaf*. The other mapping function *g* would be applied to any tree that is of type *b*, which would be *BLeaf* within the tree. This function would take these two mapping functions and the tree that the operations will be performed on and return a tree back.

This implementation of map was implemented similarly to the way I implemented the data type *AltTree* in part (a) in the sense that I viewed the tree map example shown to us in class and manipulated that code to fit this tree type. The first case that the function checks for is if a *ALeaf* is being passed as an argument. If it is an ALeaf, the function would then create a new *ALeaf* with the function *f* applied to the *ALeaf* which I called *a*. The same would be done with a *BLeaf* argument but the new leaf would have the function *g* applied to the *BLeaf* that I named *b*. Lastly, the function will need to handle if a *Node* is passed instead of *Leaf*. If a *Node* is passed then a new *Node* will be created but each argument within this new node would be a recursive call to the *altTreeMap* function. This allows for parsing through the tree that was passed as an argument until a leaf is reached. Below you can view the code I used to implement this function.

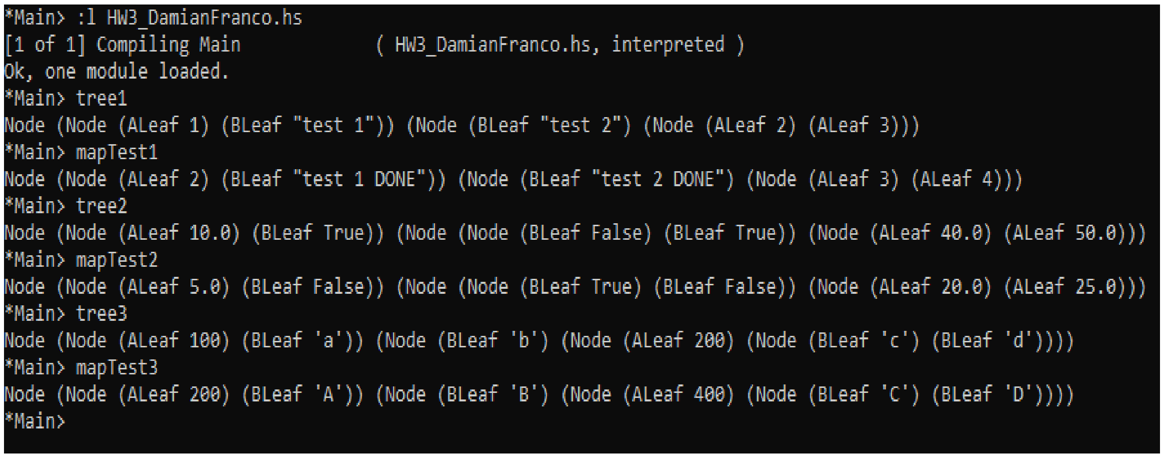


```
--Problem 3.1 (b)
altTreeMap :: (a -> c) -> (b -> d) -> AltTree a b -> AltTree c d
altTreeMap f _ (ALeaf a) = ALeaf (f a)
altTreeMap _ g (BLeaf b) = BLeaf (g b)
altTreeMap f g (Node a b) = Node (altTreeMap f g a) (altTreeMap f g b)
```

For testing I used the three trees that were initialized in part (a). The first test was to map a simple math arithmetic to all the integer values and append a string to the string values. The second test would divide each double by 2 and perform a logical negation on every boolean leaf. The last test multiplies each integer by 2 and capitalizes every character leaf. I used the *toUpper* function here from the *Data.Char* library which is why I have an import statement on the top of my Haskell file. Everything seemed to be functioning well. Below you can see the code for the tests and results of each test along with the original tree before the map function was applied.



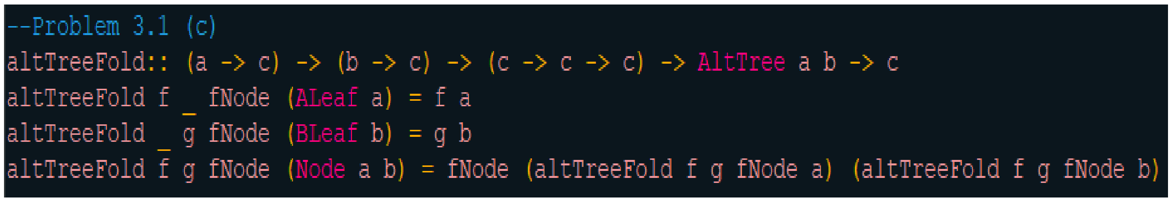
```haskell
mapTest1 = altTreeMap (\x -> x + 1) (\y -> y ++ " DONE") tree1
mapTest2 = altTreeMap (\x -> x / 2) (\y -> not y) tree2
mapTest3 = altTreeMap (\x -> x * 2) (\y -> toUpper y) tree3
```



```
*Main> :l HW3_DamianFranco.hs
[1 of 1] Compiling Main             ( HW3_DamianFranco.hs, interpreted )
Ok, one module loaded.
*Main> tree1
Node (Node (ALeaf 1) (BLeaf "test 1")) (Node (BLeaf "test 2") (Node (ALeaf 2) (ALeaf 3)))
*Main> mapTest1
Node (Node (ALeaf 2) (BLeaf "test 1 DONE")) (Node (BLeaf "test 2 DONE") (Node (ALeaf 3) (ALeaf 4)))
*Main> tree2
Node (Node (ALeaf 10.0) (BLeaf True)) (Node (Node (BLeaf False) (BLeaf True)) (Node (ALeaf 40.0) (ALeaf 50.0)))
*Main> mapTest2
Node (Node (ALeaf 5.0) (BLeaf False)) (Node (Node (BLeaf True) (BLeaf False)) (Node (ALeaf 20.0) (ALeaf 25.0)))
*Main> tree3
Node (Node (ALeaf 100) (BLeaf 'a')) (Node (BLeaf 'b') (Node (ALeaf 200) (Node (BLeaf 'c') (BLeaf 'd'))))
*Main> mapTest3
Node (Node (ALeaf 200) (BLeaf 'A')) (Node (BLeaf 'B') (Node (ALeaf 400) (Node (BLeaf 'C') (BLeaf 'D'))))
*Main>
```

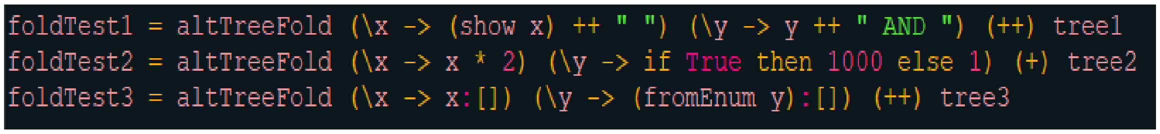
c) The last part of this question asks us to create another polymorphic function, but this time to perform folding operations over a tree rather than mapping. This function takes in two functions *f* and *g* which have the same functionality as the mapping functions in part (b), but the only difference is both *f* and *g* return the same type, type *c*, which allows for the folding operation. Just like other folding functions, another argument here will handle the main operation that will be applied to all values within the tree, which will be the third argument in this function. Lastly, an *AltTree* object will be passed as the last input to the function. The return value here will be a single aggregated value of type *c*.

Just like the previous question, the way I implemented this was by viewing multiple examples for tree folding within class notes and the internet. The structure here is very similar to the *atTreeMap* function. First, if an *ALeaf* object is passed as an argument then the function *f* will be applied to the *ALeaf* object, which I called *a*. The same will be done for a *BLeaf* object, but with the application of *g* to the object instead of *f*. My *BLeaf* argument is called *b*. Lastly, if a *Node* is passed with its argument *a* and *b* then a recursive call will be made on both *a* and *b* to parse through each branch of the tree until it reaches nodes. The other operation that this step performs is the application of the *fNode* function to each of the recursive calls. This application will generate the aggregated value that we need. Below you may view the code I wrote for this function.

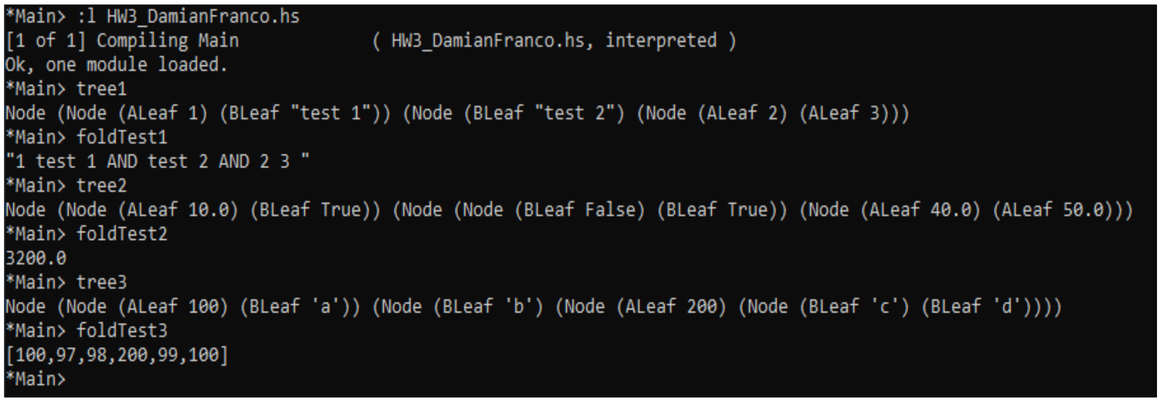


```
--Problem 3.1 (c)
altTreeFold:: (a -> c) -> (b -> c) -> (c -> c -> c) -> AltTree a b -> c
altTreeFold f _ fNode (ALeaf a) = f a
altTreeFold _ g fNode (BLeaf b) = g b
altTreeFold f g fNode (Node a b) = fNode (altTreeFold f g fNode a) (altTreeFold f g fNode b)
```

For testing this part, I did the same arithmetic as in part (b), but with a folding operation added. The first test converted every integer to a string and appended the word *AND* to every string leaf. This will then be combined into one long string because of the *fNode* function being the append call (++). Next, the second test will multiply two to every double and return 1000 if a leaf is true and 1 if a leaf is false. This would result in a double variable that would add every value together due to the *fNode* function being an add statement (+). In the last test, every integer will be put in a list and every character will be converted to an enum with the *fromEnum* function that Haskell obtains and also input in a list. This will also result in a list of integers because the *fNode* function here is also append (++). Below you may see the test code and the results of each test along with the tree input.



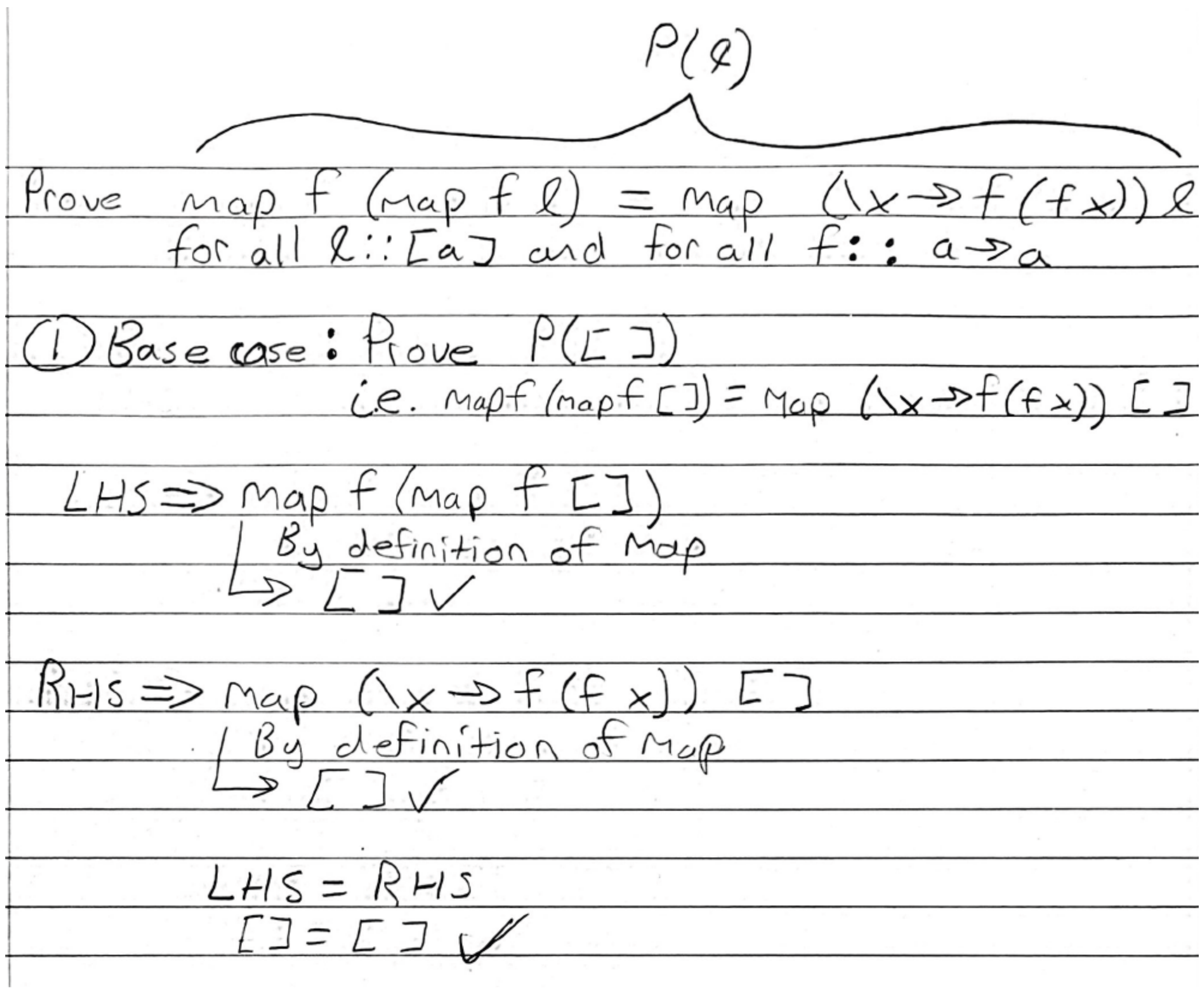
```
foldTest1 = altTreeFold (\x -> (show x) ++ " ") (\y -> y ++ " AND ") (++) tree1
foldTest2 = altTreeFold (\x -> x * 2) (\y -> if True then 1000 else 1) (+) tree2
foldTest3 = altTreeFold (\x -> x:[]) (\y -> (fromEnum y):[]) (++) tree3
```



```
*Main> :l HW3_DamianFranco.hs
[1 of 1] Compiling Main             ( HW3_DamianFranco.hs, interpreted )
Ok, one module loaded.
*Main> tree1
Node (Node (ALeaf 1) (BLeaf "test 1")) (Node (BLeaf "test 2") (Node (ALeaf 2) (ALeaf 3)))
*Main> foldTest1
"1 test 1 AND test 2 AND 2 3 "
*Main> tree2
Node (Node (ALeaf 10.0) (BLeaf True)) (Node (Node (BLeaf False) (BLeaf True)) (Node (ALeaf 40.0) (ALeaf 50.0)))
*Main> foldTest2
3200.0
*Main> tree3
Node (Node (ALeaf 100) (BLeaf 'a')) (Node (BLeaf 'b') (Node (ALeaf 200) (Node (BLeaf 'c') (BLeaf 'd'))))
*Main> foldTest3
[100,97,98,200,99,100]
*Main>
```

3.2 Proving program properties

For this question, we were given a Haskell statement to prove. That is to prove that *map f (map f l) = map (\x -> f(f x)) l*. Proofs like this are best handled by a proof by induction. That requires two steps. The two steps involve first by a base case and by an inductive case. I first started with the base case where the input list *l* is an empty list *[ ]*. I first proved that the right hand side (*RHS*) and the left hand side (*LHS*) were both equal after the empty list was the input. I found that, by the definitions of *map*, both were equal at the base case (*[ ] = [ ]*). Next, it was time to prove by the inductive case where *l* is equal to *(x:xs)*. This was much more involved than the first base case proof. I first found the inductive hypothesis (*IH*) just in case of a needed use of it to prove that the right hand side is equal to the left hand side. I started with the right hand side and used definitions to find that I could find a use for my *IH* to set the *RHS* equal to the *LHS*. All my work will be shown below.



Prove $\overbrace{\text{map f (map f l) = map (\textbackslash x -> f(f x)) l}}^{P(l)}$

for all l :: [a] and for all f :: a -> a

① Base case: Prove P([ ])

i.e. mapf (mapf [ ]) = Map (\x -> f(f x)) [ ]

LHS => map f (map f [ ])

↳ By definition of Map

[ ] ✓

RHS => map (\x -> f (f x)) [ ]

↳ By definition of Map

[ ] ✓

LHS = RHS

[ ] = [ ] ✓

② Inductive case: Prove $P(xs) \Rightarrow P(x:xs)$
for $l = (x:xs)$ for some $(x:xs)$

Assume IH: $P(xs)$
i.e. map f (map f xs) = map (\x -> f (f x)) xs

implies => map f (map f (x:xs)) = map (\x -> f (f x)) (x:xs)

RHS = map f (map f (x:xs))

By definition of map
$\hookrightarrow$ map f (f x : map f xs)

By definition of map
$\hookrightarrow$ (f (f x)) : map f (map f xs)

By IH
$\hookrightarrow$ (f (f x)) : map (\x -> f (f x)) xs

By definition of map
$\hookrightarrow$ map (\x -> f (f x)) (x:xs)

✓ DONE!
$\hookrightarrow$ LHS = RHS

map f (map f (x:xs)) = map (\x -> f (f x)) (x:xs)