Damian Franco

dfranco24@unm.edu

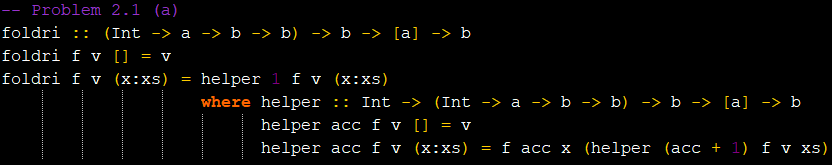
101789677

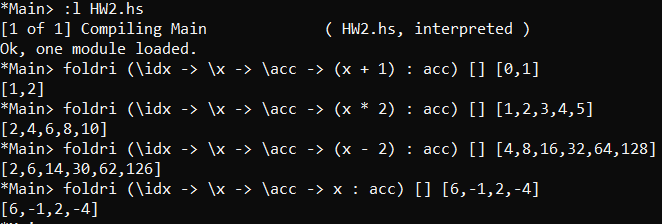
CS-558 001

Homework 2

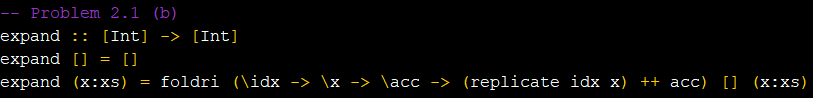
2.1 New list functionals

1. For the first question, we were asked to write a function that is based off of the *foldr* Haskell function. This function would instead need a way to track the current index in the list that is being folded upon. The higher order function that is being passed as an argument will take an extra integer argument. That integer will be the accumulator that is tracking the current index of the list.

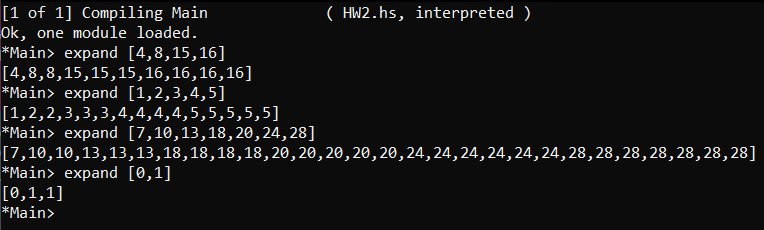
When I approached this problem, my first instinct was to utilize the *where* keyword in Haskell to create a helper function. This helper function needed to have a way of tracking an index variable. There must have also been a way of recursing through the helper function itself so the index actually keeps an accumulation variable. All arithmetic is done within this helper function. First the helper function took in an extra integer variable to track the index as well as all the arguments the *foldri* function took. Next, the function checks for an empty list and returns the *v* argument, exactly like the original foldr function definition does. If the function gets a non-empty list, the function will then apply the *f* high order function to the accumulator *acc*, the head of the list *x*, and the recursive call of the helper function as the last argument. The recursive call will then accumulate the index variable *acc* by 1 which results in a tracking of the current index of the list. Lastly, the original call of the helper function must pass through the integer 1 as the argument for *acc* due to the fact that Haskell starts the list indices at 1 instead of 0. Below you will see the code that is written. 

Below are some very basic tests to check for the functionality of the *foldri* function. This merely tested this functionality of the function and if it still works the same as foldr in certain situations. I did not find how to print out the index variable itself within the Haskell interpreter without getting a type error. Below you can see some of the tests that I used.

1. The second part of this question asks us to use the *foldri* function that we wrote in part (a) to “expand” over a list of integers. The “expand” here means that it will take a list of integers and output a new list of integers that will then duplicate the current list by the integer of the index. Meaning that if 1 is at index 1 and 3 is at index two then the output list will be [1, 3, 3].

The way that I approached this was to use a single function call of *foldri* which was the central component of my implementation. First I checked for an empty list just as a guard for empty lists. Finally the implementation calls *foldri* with the anonymous function *(\idx -> \x -> \acc -> (replicate idx x) ++ acc)*. This function does all the heavy lifting of this implementation. The *idx* variable is the input variable that tracks the index of the current element of the list. The *x* variable is the head of the list and the *acc* function is the recursive call of the helper function within *foldri* itself. The *replicate* Haskell function is utilized here with the replication of *x* for *idx* times which effectively does what the question asks us to do. There is then an append call *++* that will append the next element of the list with the current *idx* times as well. I then passed an empty list for *v* and the input list *(x:xs)* to the *foldri* function. Below you can view how I implemented this function.

Her

Testing over this function involved passing various lists with different elements and sizes. All tests seemed to function as intended. Below are some tests over the function.

2.2 Proving program properties

This question asked us to formally prove, by structural induction on lists, that *((map f) . (map g)) l = map (f . g) 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 both, by the definitions of *.* (*function composite operator)* and *map* that 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 not proceed with any other definitions without going in circles. So I then tried to do the left hand side and found that they were equal after two steps. This was by accident due to the fact of me being confused and just simply trying out the other side, but it worked out. All my work will be shown below.