Structural Inductive Proofs

Since Haskell is super cool and mathsy you can do mathematical things like proofs on functions. You can literally prove that your programs do what they should do. A structural inductive proof is very similar to a proof by induction. As ever you begin with a base case. For lists your base case will be the `[]`. Basically what you want to do is write the statement you want to prove with the `[]` inserted, then manipulate the expressions using the definitions of your functions until the LHS matches the RHS. Normally for the base case this is fairy easy.

The normal case will always be more difficult and you won't be able to get the RHS to match the LHS only using the definitions of your functions, you will also need to use the *Induction Hypothesis*. This is the statement that you are tying to prove. Often I think I'm stuck in a proof since I've run out of definitions to use but actually all I need to do is apply the *Induction Hypothesis*.

Don't worry if that doesn't make sense. It is much easier to show you through example. Lets prove the following statement `forall n > 0`:

```haskell
take n (map f xs) = map f (take n xs)
```

Intuitively this makes sense. For example if `f` was `(+2)`: adding two to a whole list, then selecting `n` elements from that list is the same as selecting `n` elements from the initial list then adding two to these.

When it comes to proofs it is always handy to have the definitions of the functions involved written close by for easy reference:

```haskell
map :: (a -> b) -> [a] -> [b]
map f []       =  []
map f (x:xs)   =  f x : (map f xs)

take :: Int -> [a] -> [a]
take _ []      =  []
take 0 _       =  []
take n (x:xs)  =  x : (take (n-1) xs)
```

First we have to prove the base case: the case in which `xs = []`

```haskell
        take n (map f [])
        === {- definition map -}
        take n []
        === {- definition take -}
        []
        === {- definition map -}
        map f []
        === {- definition take -}
        map f (take n [])
```

All that I have done is copied the statement we want to prove with `[]` instead of `xs` (`take n (map f []) = map f (take n [])`). Then I took the LHS and manipulated the expression using the definitions of `map` and `take` until I got to the LHS. So going from the first line to the second line `(map f [])` has been exchanged for `[]` following the first line in the definition of map.

Now we must prove our statement for the case `(x:xs)`:

```haskell
        take n (map f (x:xs))
        === {- definition map -}
        take n (map f (x:xs))
        === {- definition take -}
        (f x : take (n-1) (map f xs))
        === {- induction hypothesis -}
```

```
(f x : map f (take (n-1) xs))
=== {- definition map -}
map f (x : (take (n-1) xs))
=== {- definition take -}
map f (take n (x:xs))
```

This stage is done in exactly the same way except we now have an extra substitution that we can make. We can use the induction hypothesis! This means that if at any point if we have an expression in the form `(take n (map f xs))` we can swap it our for `(map f (take n xs))` or vice versa, in the example proof the IH happens when `n = (n-1)` which is perfectly valid. Clearly we couldn't have used this immediately due to the presence of the `x` before the `xs`. Since we have now isolated the smaller case of `xs` we can use the IH because induction is based on assuming something for a smaller case making the bigger case true.

Proofs are a game of applying definitions in the right way so that the RHS equals the LHS, and when there is an induction hypothesis involved I just normally apply definitions to make it look sort of similar then normally you will find that all you have to do is apply the induction hypothesis and you're done!