Dissertation log:

* Georg mentioned that I would need an ATP system. But it seems that I would end up creating this anyway during the programming of this project. So there’s no need to find an external one to use. This is the case that is used with other program verification tools as well (Stanford Pascal Verifier)
* According to Georgs lecture, the actual syntax/grammar was done using a tree data type. But I found out that it is enough to use a simple data type, but with some recursion involved for specific cases
  + This is Georgs grammar for and, or, implies propositional variables:
    - Φ ::= ⊥ | ⊤ | p | (¬ Φ) | (Φ ∧ Φ) | (Φ ∨ Φ) | (Φ → Φ)
  + The grammar I’m using is on arithmetic expressions like +, -
  + I’ve implemented +, -, x but haven’t done division yet as I’ve not got around to figuring out how to divide using natural numbers, and I don’t know if it’s even possible
  + here’s the data type:
    - data Expr = Val Nat
    - | Var Variable
    - | Mult Expr Expr
    - | Add Expr Expr
    - | Sub Expr Expr
    - deriving (Eq, Ord, Show, Read)
  + As you can see, the function is recursive in the sense that you can use an expression as an argument to one of the possible expressions. This is why I need Val Nat or Var Variable, because that’s the base value where there’s no more recursion.
* I’ve implemented the natural data types and 3 arithmetic functions that make use of the datatype:
  + data Nat = Zero | Succ Nat
  + deriving (Eq, Ord, Show, Read)
  + add :: Nat -> Nat -> Nat
  + add Zero n = n
  + add (Succ m) n = Succ (add m n)
  + mul :: Nat -> Nat -> Nat
  + mul Zero n = Zero
  + mul (Succ m) n = add (mul n m) n
  + sub :: Nat -> Nat -> Nat
  + sub n Zero = n
  + sub Zero (Succ Zero) = Zero
  + sub (Succ n) (Succ m) = sub n m
* When using natural numbers, the numbers are often extremely long since I need to include the successions. For example 3 would be Succ(Succ(Succ Zero)). I’m trying to figure out how to make it so that I can input integers and have them transferred to natural numbers, but because of the complexity of the functions that I’ve written later on, it proved to be quite a complicated matter. I’m still in the process of finding out if its impossible or not, since it would really improve the appearance of commands and readability.
* To make my code a bit fancier, I’ve used a control stack to perform my arithmetic. I needed a type:
  + type Cont = [Op]
* and I needed a datatype Op
  + data Op = EVALMULT Expr
  + | EVALADD Expr
  + | EVALSUB Expr
  + | MULT Nat
  + | ADD Nat
  + | SUB Nat
* The abstract machine uses eval and exec functions to basically control the order of evaluation. For example, when doing something like value (Add (Add (Val 2) (Val 3)) (Val 4)), which is basically (2+3)+4 Haskell would follow a default order of evaluation, so it might evaluate Add ‘(X X) (Val 4)’ first instead of Add (Val 2) (Val 3). An abstract function assigns an order
  + It does this by making use of a control stack
  + The abstract machine uses two mutually recursive functions, eval and exec.
  + There’s more information in Graham Hutton’s book on Haskell where I got this idea from, If I need to explain the control stack in more detail I can refer to this book.
  + For now, know that there is a sequential composition that takes place, and in my abstract machine, it is from left to right, in the order it is written
  + The ‘value’ function uses this abstract machine to give me my final output in natural numbers
* I included a validation function that basically checks if my nat arithmetic functions work properly (comparing my add function to +: (Add (Val 2) (Val 3)) == Val 2 + Val 3) but that seemed to be pretty useless, so I deleted it
* I have nat2int and int2nat functions that ive been planning to use to help me change my long nat input to a shorter one, but that’s a work in progress.
* I have a datatype for statements. These are inequalities such as less than and equal to. I’ve basically added the whole list with equal variants (MoreEqual), but apparently, all that is needed is less than and equal according to Georg. I can make the rest using those two, but for now im working with this:
  + data Statement = Less Expr Expr
  + | LessEqual Expr Expr
  + | Equal Expr Expr
  + | More Expr Expr
  + | MoreEqual Expr Expr
  + | F
  + | T
* ‘T’ and ‘F’ being true and false, because using ‘True’ and ‘False’ doesn’t work since Haskell gets confused between my data type and Haskell’s Boolean expressions.
* The ‘evalStatement’ function basically gives meaning to the datatypes by checking the actual inequalities. Whenever I need a statement checked, I run it through this function.
  + evalStatement :: Statement -> Store -> Bool
  + evalStatement (Less w z) s = (value w s) < (value z s)
  + evalStatement (LessEqual w z) s = (value w s) < (value z s) || (value w s) == (value z s)
  + evalStatement (Equal w z) s = (value w s) == (value z s)
  + evalStatement (More w z) s = (value w s) > (value z s)
  + evalStatement (MoreEqual w z) s = (value w s) > (value z s) || (value w s) == (value z s)
  + evalStatement F \_ = False
  + evalStatement T \_ = True
* I have also introduced a store. Now the store is important because we need a place to store the variables. Now that I am mentioning it, variables were also needed in this case, so:
  + The variables datatype has been added:
    - data Variable = U | V | W | X | Y | Z
    - deriving (Eq, Ord, Show, Read)
  + And variables can now also be used in expressions as ‘Var Variable’
  + Now the fact that variables are included as expressions means that variables also need to be able to hold a number. As you can see in the abstract machine, when Var is detected as an expression, it leads to the function ‘find’. This is part of the set of functions made for manipulating the store.
  + The store consists of variables and expressions, the type of the store is:
    - type Store = [(Variable,Expr)]
  + The store looks like this:
    - store :: Store
    - store = [(U, Val (Succ Zero)),
    - (V, Val (Zero)),
    - (W, Val (Succ (Succ (Succ Zero)))),
    - (X, Val (Succ (Succ Zero))),
    - (Y, Val (Succ Zero)),
    - (Z, Val (Zero))]
  + And the functions are:
    - find :: Variable -> Store -> Expr
    - find v s = head [n | (v',n) <- s, v == v']
    - delete :: Variable -> Store -> Store
    - delete v s = [(v',n') | (v',n') <- s, v' /= v]
    - update :: Variable -> Store -> Expr -> Store
    - update \_ [] n = []
    - update v (x:xs) n
    - | (fst x) == v = (v,n):xs
    - | otherwise = x:update v xs n
  + Delete hasn’t got a use for now, but it seemed right to include it as one of the important list manipulating functions that are usually included in Haskell programs
  + Find returns the corresponding expression that is assigned to the variable, as it shows.
  + Update returns an entirely new store after that expression of the chosen variable has been changed. This is where the uniqueness of Haskell comes into play.
  + Haskell doesn’t let you update lists. Once a list is initialized you cannot change it while the code is running. Instead you can create new lists and use those instead, hence the update function returning an entirely new list for every update that it makes.
  + This also means however, that the store has to be passed to every function as an argument, as this is information that we need and cant be stored; I’ve not found a possible way to store new lists to a new variable and I don’t think its possible in Haskell.
  + You’ll notice as a results that almost every function that has anything to do with manipulating numbers requires the store as an input. This is so that we can make use of the variables, which is an important factor for us.
  + The store lets us to stuff like ‘X + V’
* Now we move into the programming language that our commands will be written in:
  + This is the data type for now
    - data Lang = Assign Variable Expr
    - | If Statement Lang Lang
    - | While Statement Lang
    - | Order Lang Lang
  + The while statement can be left out for now, since it adds an extra level of complexity that isn’t ideal for this project (introduces problem of soundness and completeness)
  + Each language command has its own function:
    - inter :: Lang -> Store -> Store
    - inter (Assign v e) s = update v s (Val (value e s))
    - inter (If x l1 l2) s
    - | evalStatement x s = inter l1 s
    - | otherwise = inter l2 s
    - inter (While x l) s
    - | evalStatement x s = inter (While x l) s'
    - | otherwise = s
    - where s' = inter l s
    - inter (Order l1 l2) s = inter l2 s'
    - where s' = inter l1 s
  + Where inter = interpret.
  + The assign function does us the favour of actually evaluating the expression that you wish to assign to a variable. So if you chose to assign (5+6\*23) to X, the function will actually calculate that using ‘value’ then “store” it (return a new store) in the variable you have chosen.
  + The interpret function returns a store because the store is basically where everything is going to be compared. The ‘evalStatement’ function is going to use the store to check if the pre and post conditions are true.
  + The if statement has two ‘Lang’ arguments to allow for more than one if statement (if elif elif elif else). But the problem with this is the fact that you have to call ‘If’ as the second lang, so it will end up as a recursive if function which is very long. However, I’m going to leave it this way as its not expected that any more than 2 if functions are going to be used.
  + Similarly, for the ‘Order’ language command I have included to emulate sequential composition, the same problem exists. With ‘Order’ it’s a bigger problem because it’s more likely that the user wishes to include a bunch of commands compared to a bunch of if statements.
  + Long sequenced commands will therefore look like this:
    - Order (Assign (Z) (Add (Var Z) (Val (Succ Zero)))) (Order (Assign (Y) (Add (Var Y) (Var Z))) (Assign (V) (Add (Var V) (Var Y))))
  + You can see why I’m trying to shorten the input.
  + I plan to change this too, or at least find a different way to implement sequential composition. Possibly with the use of an abstract machine.
* Georg has explained to me that my language commands ‘Assign’, ‘If’ etc. are actually syntactic commands, and the function ‘inter’ that interprets each syntactic command is the semantic function behind it.
* Semantics, after all, are the explanations of the syntax that is used. And I’ve managed to implement it in this way.
* Without my ‘inter’ function, my language commands ‘Assign’, etc. will mean nothing. But with this semantic function it allows it to actually perform its desired functions.
* I’ve also tried to implement a ‘triple’ function:
  + triple :: Statement -> Statement -> Store -> Lang -> Bool
  + triple p q s l
  + | evalStatement p s = evalStatement p s'
  + | otherwise = False
  + where s' = inter l s
* But Georg advises that I should leave that till all the prerequisites are done first.
* Georg also advises that I look at how Mike Gordon codes his semantics