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
  + n\_div :: Nat -> Nat -> Nat -> Nat
  + n\_div \_ Zero \_ = Zero
  + n\_div n f c
  + | (d\_sub n f c == Zero) = add (c) (Succ Zero)
  + | otherwise = n\_div n' f (add (c) (Succ Zero))
  + where n' = d\_sub n f c
  + d\_sub :: Nat -> Nat -> Nat -> Nat
  + d\_sub Zero \_ \_ = Zero
  + d\_sub n Zero \_ = n
  + d\_sub (Succ n) (Succ m) c = d\_sub n m c
* 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 result 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 do stuff like ‘X + V’
* Now we move into the p rogramming 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
* **Infix Operators to Shorten Data Elements**
  + Attempted to shorten the datatype elements by introducing infix operators in Haskell. This calls for the need of a GHC extension XTypeOperators. But have had no success implementing them yet as I’m not sure it’s possible to use and infix operator as an element name (Assign or Add)
* **Sequential Composition**
  + Sequential composition is fine as Georg says. This is how its implemented, but it needs to go from right to left. Because Hoare logic runs the command by evaluating the post condition first then proving that the precondition can be reached through the reverse of the command.
  + If statements might not work in sequential composition if it is moving backwards like this, but further testing and explanation is needed
    - Possible question to ask Georg
* **Division of Natural Numbers Is Possible**
  + Division of natural numbers is possible through heuristic algorithms. It’s done by subtracting the number by the factor and seeing how many times it can subtract before it reaches the remainder. With natural numbers you can just discard the remainder and use the counter for how many times you were able to subtract by the factor.
    - Division is not as necessary at the moment but will be a nice addition to the set of expressions.
* **Hoare Logic and Number of Stores**
  + Normally Hoare triples work over all possible stores. This is what is normally referred to as complete Hoare logic, because it works over all stores
  + In our project, we will limit the number of stores to a specific number. Possibly up to 100. And consider these stores up to the limit ‘All stores’
  + If we would want to go beyond the limit of stores we have set, we would have to use and external SMT Solver which would perform the arithmetic for the Hoare triple command over all possible stores.
  + Mike Gordon uses ‘state’ to refer to the store, which is the same thing as it is passed through all ESems (Expression semantics), SSems (Statement semantics) and HSems (Hoare-triple semantics) that he implements
  + The stores will all have the same variables, but each with a different set of values for each variable. This is so that we can test if the Hoare triple works over a large set of values.
  + The number of variables doesn’t have to be large (Can be 3/4)
  + A possible method of creating a set of stores is by using 2D arrays. Basically, having a list of stores and iterating and applying the Hoare triple command over each store then checking if the result of each matches the Precondition statement (Because we are going backwards from Q to P)
    - A suitable command for this project would be the variable substitution command, where two variables swap their values
    - This would require 3 variables, and a sequence of commands to implement
* **Swap command and Triple over all stores implementation**
  + I’ve created a list of stores called stores:
    - type Stores = [Store]
    - stores :: Stores
    - stores = [store,store2]
  + I’ve also written the language command that will swap two variables, X and Y. The values of X and Y will be stored in Z and V respectively.
    - swap\_vars :: Lang
    - swap\_vars = Seq (Assign (Y) (Var Z)) (Seq (Assign (X) (Var Y)) (Seq (Assign (V) (Var Y)) (Assign (Z) (Var X))))
  + And finally the ‘triple\_recursion’ function that takes a list of stores (type Stores) and checks the Hoare triple over all the stores:
    - triple\_recursion :: Statement -> Statement -> Stores -> Lang -> Bool
    - triple\_recursion p q (x:xs) l
    - | triple p q [] l = True
    - | triple p q x l = triple\_recursion p q xs l
    - | otherwise = False
  + Problems:
    - I need to know exactly what to use as pre and post conditions. For now, I’ve used ‘True’ for the precondition and ‘X==V’ for the post condition.
    - Which leads to the second problem of the fact that I can only have one condition (Statement) at a time. So I can only write ‘X==V’ and not ‘Y==Z’ as well.
    - Another problem is the question of whether I need to assign every single store from 1-100. I’m assuming just proving that the algorithm works for a few stores will prove its ability to work over all, but I need confirmation.
* **Negation and Conjunction (Multiple statements)**
  + Georg says that along with the <, = statements, I should have a conjunction (and) and a negation element in my statements datatype.
  + I have implemented the negation using ‘Not Statement’ Which simply returns false if the statement is true and vice versa
  + I have implemented conjunction using a list of statements. So instead of having a long recursive statement similar to my sequential composition implementation, I have made it so that the triple function works with lists of statements for pre and post conditions.
  + However this isn’t enough as I still need ‘And’ for statements in ‘If’
  + Meaning the triple function iterates over all pre conditions of a single hoare triple and checks them with the current store before moving on to the command and executing it, followed by checking the new store with all the post conditions before returning a result
  + This is the code for now
    - triple :: Statements -> Statements -> Store -> Lang -> Bool
    - triple [] q s l = evalPost q s'
    - where s' = inter l s
    - triple p@(x:xs) q s l
    - |evalStatement x s = triple xs q s l
    - |otherwise = False
      * If the pre conditions are empty, the run the command and check the post conditions using ‘evalPost’
      * Otherwise, use Haskell recursion to evaluate all the precondition until there are no more, once it’s empty it goes to the previous line
    - evalPost :: Statements -> Store -> Bool
    - evalPost [] s = True
    - evalPost (x:xs) s
    - |evalStatement x s = evalPost xs s
    - |otherwise = error "Post condition could not be reached"
      * If there are no post conditions, return True
      * Otherwise use Haskell recursion to evaluate all post conditions in the list until there are no more, once its empty it goes to the previous line
      * If the evaluation of a post condition returns false, then an exception is raised
    - triple\_recursion :: Statements -> Statements -> Stores -> Lang -> Bool
    - triple\_recursion p q [] l = True
    - triple\_recursion p q (x:xs) l
    - | triple p q x l = triple\_recursion p q xs l
    - | otherwise = triple\_recursion p q xs l
      * This functions allows multiple stores to be used. Basically calls ‘triple’ over multiple stores
      * If the stores are empty, return True
      * Otherwise use Haskell recursion to call ‘triple’ over all the stores.
      * If ‘triple’ returns false, then just continue with the rest of the stores, as an exception would have been raised anyway if it was false, and all triples with preconditions that don’t hold will be ignored.
* **Pre and Post statements**
  + When it comes to the swap variables command, the pre and post conditions have to be fixed to specific values. So you would need to say PRE: {X:= 5, Y:= 6}, POST: {X:= 6, Y:= 5}.
  + The problem that comes with this is that it would only work for one store out of the finite stores we have.
  + The precondition doesn’t hold in the rest of the stores therefore we cant check if the hoare triple is true.
* **What to do when pre condition doesn’t hold?**
  + According to Georg, any store that doesn’t satisfy the precondition can be ignored. And the next store is checked instead. This is because if the pre condition is false, the hoare triple cant be proven
  + If the post condition doesn’t hold, then we raise an exception. We don’t label it as false because there can be many factors affecting the result of the command. Either the command is faulty or the post condition is wrong or whatever
  + The only result we can get is true which is if it passes all stores without an exception raised.
  + Heres a truth table for Pre -> Post:
    - T -> T = T
    - F -> T = Ignore
    - T -> F = Exception raised
    - F -> F = Ignore
* **How many stores to use?**
  + Georg says that the idea is that if I can make my functions work when I pass in 2 stores, then it will work if I pass 10 stores, and so on. The importance is not the amount of stores, but the actual workings of the hoare logic that im trying to implement
* **Division command**
  + I have implemented a command that divides 2 numbers:
    - {Y > 0, X > Y} V:= Div X Y {X == V \* Y}
  + However Georg says this is too simple, and suggests implementing the same thing but using a while loop instead
  + Turns out the function is faulty. Since the division doesn’t consider decimals, the result is always a whole number, so some info is lost.
  + Example:
    - 21 / 5 = 4.2
    - But ‘Div 21 5’ gives us 4
    - Hence 4 \* 5 != 21, so exception is thrown
* **Division Using While Loop:**
  + I’ve implemented division using a while loop and if statement all in one command.
  + The actual command is too long, so heres the concise version in order of execution
    - PRE: {Y > 0, X > Y}
    - 1. Z := X, V := 0
    - 2. While (Z >= Y): {Z := Z – Y, V := V + 1}
    - 3. If (Z != 0 && Z < Y): {U := X} Else: {U := 0}
    - POST: {X == (V\*Y) + U}
  + This method ensures that the remainder is preserved for accurate equality between the original value and the value multiplied back
* **Symbolic Constants/Variables**
  + To eliminate the problem of infinite stores, the idea of symbolic variables are introduced
  + These variables have no values associated with them and are simply used to act as placeholders for possible numbers
  + The idea is to use them in hoare triples to prove that the command could work for any number as represented by the symbolic variables
  + This eliminates the need of stores. Since if the triple holds symbolically, then it should hold for all stores.
  + A function is needed to simplify expressions including symbolic variables into their ‘normal forms’ which is the form once you’ve simplified the expression as much as possible. For example:
    - m + 8 + 3 + n
    - m + n + 11

TODO (in order of least to most important):

* Think of a way to shorten commands
* Check if ‘IF’ statements work with backward sequential composition
* Implement conjunction in statements
* Make while division
* Implement symbolic constants/variables