WEEK08 Presentation

AGENDA:

1. Quick Assignment 2 take up
2. Quick Exercises05 take up
3. Quick context week 08 - correctness proof
4. Group practice problems to help with Exercise06! - competition ?
5. Explain in detail Exercise06 (Hints)
6. Explain problems with Assignment 2

Reasoning about programs.

- How do we understand fundamentally what a function does?
    - We evaluate particular inputs in GHCI
    - Do the same thing by hand and manually calculate
    - We can REASON about how the program behaves in general
        - Can do this by defining a property this certain function holds

```
sum :: Num a => [a] -> a
sum [] = 0                ---def sum.1
sum (x:xs) = x + sum xs.  -- def sum.2


sumOneProp :: Integer -> Bool
sumOneProp x = sum[x] == x
```

Proof for the case above:

```
sum [x]
= sum(x:[]).
= x + sum[]      ---by def sum.2.
= x + 0          ---by def sum.1.
= x              ---Integer arithmetic (to avoid floating point errors)
```

DEFINEDNESS & TERMINATION

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n* fact (n-1)
```

Fact (-1) --- what happens? NOT DEFINED!!

```
neverEnd n = if n>= 0 then n else neverEnd (n-1)
                                                  --- For what numbers is it "not defined?"?
mod :: Integral a => a->a->a
mod x y = x - z * y
where z = x `div` y
-- for what numbers is it not defined?
-- does it still terminate ? What happens?
```

These equations don't hold for anywhere, where it doesn't terminate or it isn't defined.

LOGIC

"logical if then else"
"if a is true then b must also be true"

Implication (=>) A=> B

- "A implies B"

```
a ==> b = case(a,b) of
(True, False) -> False
_ -> True
```

Logical implication [edit]

Logical implication and the material conditional are both associated with an operation on two logical values, typically the values of two propositions, which produces a value of *false* if the first operand is true and the second operand is false, and a value of *true* otherwise.

The truth table associated with the logical implication **p implies q** (symbolized as **p ⇒ q**, or more rarely **Cpq**) is as follows:

Logical implication

p	*q*	*p* ⇒ *q*
T	T	T
T	F	F
F	T	T
F	F	T

Logical implication is denoted "P implies Q"

- If you start with P being false, then you are free to conclude what you wish (this is the second of half of the truth table).
- If P is true, the implication should be true only if Q is true.

An example:

"Get an *A* in your exam, then I will buy you a car."

i.e. you are promising a Kid a car IF they get an A in their exam.

Where
p = Kid get's an A in exam
q =Bought kid the car
i.e.

For p = true

- **if kid gets an A in exam (p=true)** AND **you bought him a car (q = true**), THEN the promise was held

(P=>Q = True)

- **if kid gets an A in exam (p=true)** AND **you did not bought him a car (q = false**), THEN, then the promise didn't hold and p=>q is false

For p= false

- **If kid DOES NOT get an A in exam (p = false)** and **you bought him a car (q = true)**, THEN the promise was "technically" still held. You only said what would happen if the kid get's an A, never what will happen if kid does not get an A. (p=>q is True) (i.e. this statement is not IF AND ONLY IF)
- **If kid DOES NOT get an A in exam (p = false)** and **you did not buy him a car (q = false)**, THEN your promise still holds, since you only guaranteed the kid a car if he gets an A, which he didn't ; therefore you did not buy him a car (p=> q True)

AND, OR

Logical conjunction (AND) [edit]

Logical conjunction is an operation on two logical values, typically the values of two propositions, that produces a value of *true* if both of its operands are true.

The truth table for **p AND q** (also written as **p** ∧ **q**, **Kpq**, **p & q**, or **p** · **q**) is as follows:

Logical conjunction

p	*q*	*p* ∧ *q*
T	T	T
T	F	F
F	T	F
F	F	F

In ordinary language terms, if both *p* and *q* are true, then the conjunction *p* ∧ *q* is true. For all other assignments of logical values to *p* and to *q* the conjunction *p* ∧ *q* is false.

It can also be said that if *p*, then *p* ∧ *q* is *q*, otherwise *p* ∧ *q* is *p*.

Logical disjunction (OR) [edit]

Logical disjunction is an operation on two logical values, typically the values of two propositions, that produces a value of *true* if at least one of its operands is true.

The truth table for **p OR q** (also written as **p** ∨ **q**, **Apq**, **p || q**, or **p + q**) is as follows:

Logical disjunction

p	*q*	*p* ∨ *q*
T	T	T
T	F	T
F	T	T
F	F	F

TESTING PROPERTIES WITH ASSUMPTIONS

```
factProp :: Int -> Bool
factProp n = fact n `div` n == fact(n-1)
-- what will happen when we call quickcheck on this?
```

Proof factProp holds when defined ! Do together!!

Fact 0 = 1
Fact n = n * fact(n-1)

Fact(n) 'div' n

....

=fact(n-1)

Structural Induction On Lists

- Proving a property P holds for all finite lists 'xs' i.e. P(xs) holds for all lists.

1. Base Case ; Prove P([])
2. Induction step; Prove P(xs) => P(x:xs) (i.e. prove the property holds when an element is added to the list

P(xs) is known as 'Induction Hypothesis' (WE ASSUME THIS TO BE TRUE)

THEREFORE two things can happen if we recall logical implication:

A =P(xs) ; always assuming to be true
B = P(x:xs) ; either true or false

A => (B = FALSE) THEREFORE A => B = FALSE
A => (TRUE = TRUE) THEREFORE A => B = TRUE

```
reverse :: [a] -> [a]
reverse [] = []. (1)
reverse (x:xs) = reverse xs ++ [x]  (2)


reverseProp2 :: [Integer] -> Bool
reverseProp2 xs == reverse(reverse xs) == xs  ==> TRUE



   1. BASE CASE

reverseProp1([]) = reverse(reverse [])
                 = reverse ([]) by (1)
                 = []           by (1)
                 --------TRUE FOR BASE CASE

   2. INDUCTION STEP:

           Lemma:  reverse(xs ++ ys) == reverse ys ++ reverse xs  -- assume to be true
ReverseProp2(xs) ---=> Assume to be true
              ReverseProp2(xs) = reverse(reverse xs) = x=s <- Induction Hypothesis

Prove for:

                                  Reverse(Reverse(x:xs)) = x:xs


ReverseProp(x:xs) = reverse(reverse(x:xs))
                  = reverse (reverse xs ++ [x])  -- by (2)
                  = reverse ([x]) ++ reverse (reverse xs)  --by lemma
                  = reverse (x :[]) ++ xs. -- by induction hypothesis
                  = reverse [] ++ [x] ++ xs
                  = x:xs
```

GROUP PROBLEM::

```
reverse :: [a] -> [a]
reverse [] = []. (1)
reverse (x:xs) = reverse xs ++ [x]  (2)
```

1. SOLVE LEMMA ABOVE BY INDUCTION HYPOTHESIS! P (x:xs ++ ys)
2. Solve the sumProp2 property below:

```
sum :: Num a => [a] -> a
sum []     = 0          -- def sum.1
sum (x:xs) = x + sum xs -- def sum.2
```

Prove the following property

```
sumProp2 :: ([Integer],[Integer]) -> Bool
sumProp2 (xs,ys) =  sum (xs ++ ys) == sum xs + sum ys
```

You're allowed to use the following

```
-- Lemma
    x : (xs ++ ys) == (x:xs ++ ys)
```

3. Solve takeLengthProp :: [a] -> Bool
   take LengthProp xs = take (length xs) xs == xs

Using:

Length:: [a] -> int
Length [] = 0
Length (x:xs) = 1+ length xs

Take _ [] = []
Take n (x:xs)
| n> 0 = x : take (n-1) xs
| otherwise = []

** NEXT PAGE FOR SOLUTIONS **

SOLUTION FOR LEMMA

```
reverse :: [a] -> [a]
reverse [] = []. (1)
reverse (x:xs) = reverse xs ++ [x]  (2)
```

```
LEMMA:  reverse(xs ++ ys) == reverse ys ++ reverse xs  -- assume to be true
```

Base Case:

```
Reverse([] ++[]) = reverse [] ++ reverse []
Reverse [] = [] ++ []
[] = [] TRUE
```

Induction hypothesis:

```
ASSUME TRUE: reverse(xs ++ ys) == reverse ys ++ reverse xs

PROVE FOR: P(x:xs) => reverse(x : (xs ++ ys)) == reverse ys ++ reverse x:xs

        1. Reverse(x:xs ++ ys)
     Reverse (x : (xs++ys)
     = reverse(Xs++ys) ++ [x]           --- by(2)
     = reverse ys ++ reverse xs ++ [x]  -- induction hypothesis
     = reverse ys ++ revese(x:xs)       --by (2)
```

Solution for Sum

```
sum :: Num a => [a] -> a
sum []     = 0          -- def sum.1
sum (x:xs) = x + sum xs -- def sum.2
```

Prove the following property

```
sumProp2 :: ([Integer],[Integer]) -> Bool
sumProp2 (xs,ys) =  sum (xs ++ ys) == sum xs + sum ys
```

You're allowed to use the following

```
-- Lemma
    x : (xs ++ ys) == (x:xs ++ ys)
```

Base Case:

```
Sum ([] ++ []) = Sum [] + Sum []
Sum ([]) = Sum [] + Sum []
0 = 0 + 0
0 = 0  TRUE
```

Induction Hypothesis

```
Assume true: sumProp(xs,ys) = sum(xs++ys) = sum xs + sum ys

Prove P(x:xs) i.e.  sumProp(x:xs,ys) = sum(x:xs++ys) = sum x:xs + sum ys

Sum(x:(xs++ys)
=x + sum(xs++ys)                    -- def sum.2
= x + sum xs + sum ys               --- induction hypothesis
= sum(x:xs) + sum ys                -- def sum.2
```

Solution for takelengthProp

takeLengthProp xs = take (length xs) xs == xs

```
Length:: [a] -> int
Length [] = 0
Length (x:xs) = 1+ length xs.  --take.2

Take _ [] = []   take.1
Take n (x:xs)
      | n> 0 = x : take (n-1) xs --take.2
      | otherwise = []
```

Base case

```
Take(length[]) [] = [] -- by take.1
```

Induction Hypothesis

```
ASSUME TRUE: take(length xs) xs == xs

PROVE FOR P(x:xs) => take(length (x:xs) (x:xs) = x:xs

        1. Take(length(x:xs)) (x:xs)
     = take (length xs + 1) (x:xs)   --- by length.2
     = x: take (length(xs) xs)       -- by take.2
     = x: xs
```