FMFP THINGS TO KNOW

FP PART

Typing Expressions

- only give type for what is unknown, i. e. :

```
map (: 1) :: [[a]] -> [[a]]
-- and not :
map (: 1) :: (a -> a) -> [[a]] -> [[a]]
-- since we already have the function (: 1) given, it is not
unknown, no need to type it.
```

- respect parenthesis, i.e. :

```
(\f -> (\h -> (f,h))) :: a -> (b -> (a, b))
```

- if the expression is an if-statement, be careful that the two possibilities for the return have the same type, i.e. :

```
\x y z -> if z x then y x else x :: a -> (a -> a) -> (a -> Bool) ->
a
```

Typing Proof in Mini-haskell:

- when the first rule application is the two-term side by side rule $((\ldots) :: t_0)$ then have to go up until the next rule application is *Abs*

Defining the Fold function on Trees:

- type : (a->b) -> (a ((only if Node takes an argument a else no)) -> n times b after that depending on the number of Trees in the definition -> then once Tree a again -> b
- function definition :

```
foldTree l n = go
    where go
        go (Leaf a) = l a
        go (Node a t1 t2) = n a (go t1) (go t2)
```

- mapTree definition:

```
mapTree f tree = foldTree leaf node tree where
  leaf x = Leaf (f x)
  node x t1 t2 = Node (f x) t1 t2
```

Instantiate the type class Eq for an object:

```
instance Eq a => Eq (Object) where
    (==) (type1 a) (type1 b) = a == b
    (==) (type2 l1) (type2 l2) = all (\e -> e `elem` l2) l1 && all (\e ->
e `elem` l1) l2
    (==) _ _ = False
```

Induction Proof

- If the lemma has a number in it (i.e. 0) then first prove the general lemma by replacing the number by *n*, then you can use the general lemma to prove the particular case. Same with an empty list, first replace by normal list ys.
- Example:

```
Lemma: (take n xs) ++ (drop n xs) .=. xs

Proof by induction on List xs generalizing n

Case []
    For fixed n
    Show: (take n []) ++ (drop n []) .=. []
    Proof
    QED

Case x:xs
```

```
    Fix x, xs
    Assume
        IH: forall n: (take n xs) ++ (drop n xs) .=. xs
    Then for fixed n Show: (take n (x:xs)) ++ (drop n (x:xs)) .=. (x:xs)
    Proof

    QED
QE
```

Haskell Programming

Most Important Prelude Functions

- scanl : takes function, start element and array, and applies function to element and 1st of array ... and returns array of results.
- foldl: same as scanl, only that it returns single value not an array : result of function application to all elements together

```
map f (x:xs) = f x : map f xs
```

- Filter: remove elements of xs that do not return true when the funciton is applied to them and return resulting array
- Concat: turn array of arrays to single array
- Iterate: creates infinite list where the first item is calculated by appliying the function to second arg...
- Take: take the n first elements of list
- Drop: drop n first elements of list
- Zip: takes two lists and returns list of tuples of one element of each list
- Zip with: takes a function and two lists, returns a list of the function applied to one of each list.
- Takewhile : only takes n first elements that return true when the function is applied to them
- dropwhile : “”
- Until: applies function to the 3rd arg and it compares the result with the condition, if the condition returns true it prints the result, else it passes the result to the function and repeats the cycle.

- Sort: sorts list
- Maybe Type : Nothing / Just x
- Div: (same for mod)

```
(number to divide) `div`(number to divide by)
```

=> mod returns the remainder, div returns the result of the division.

Show Instantiation

```
instance (Show a) => Show (F a)
  where
```

After just use the fold function defined earlier and gice corresponding arguments to print the correct strings.

FM PART

Small Steps Semantics

- $$\begin{aligned}\langle s, \sigma\rangle &\rightarrow_1 \langle \text{if } x > 0 \text{ then } x := 3 - 2 * x;\ s \text{ else } skip \text{ end}, \sigma\rangle \\ &\rightarrow_1 \langle x := 3 - 2 * x; s\ , \sigma\rangle \\ &\rightarrow_1 \langle s, \sigma[x \rightarrow 1]\rangle\end{aligned}$$

$$\frac{\frac{\overline{\langle x:=3,\sigma\rangle \rightarrow_1 \sigma[x \mapsto 3]}\, ASS_{SOS}}{\langle x:=3;y:=4,\sigma\rangle \rightarrow_1 \langle y:=4,\sigma[x \mapsto 3]\rangle}\, SEQ1_{SOS}}{\langle (x := 3; y := 4); s_{loop}, \sigma\rangle \rightarrow_1 \langle y := 4; s_{loop}, \sigma[x \mapsto 3]\rangle}\, SEQ2_{SOS}$$

- **Semantic Equivalence** = s_1, s_2 are semantically equivalent $\equiv$ $\forall \sigma, \sigma'. (\vdash < s_1, /sigma > \rightarrow \sigma' \Leftrightarrow < s_2, \sigma > \rightarrow \sigma')$

=> Same for Big Step Semantics

Big-Step Semantics Proof (While Property)

- Define the predicate : $P(T) \equiv$ (the property that we need to prove) "*and prove* $\forall T.\ P(T)$ *by strong structural induction over the shape of the derivation tree T using the induction hypothesis* $\forall T' \sqsubset T.\ P(T')$".
- Let the used states $\sigma \ldots$ be arbitrary, the conditions of the LHS of the property are to be assumed. The last rule applied in T must be a while rule, hence we have to distinguish two cases over the condition b of the while rule.
- Case WHF_{NS}:
  - With the false while rule being the last rule applied we must have $B[b] = ff$.
  - Then write False rule for the LHS, say $\sigma = \sigma'$, *We can now construct a derivation tree* T' *with* $root(T') \equiv \ldots$ *using* $\sigma = \sigma'$ *that looks as follows...*
- Case WHT_{NS}:
  - $B[b] = tt$
  - *hence the derivation tree T with root(T)... looks as follows* : (use while true rule and then eventually develop another rule because of the statement inside the while loop)
  - Then end with T_1, T_2 (or more/less depending on the case) for some state σ_1
  - *Then we can construct a derivation tree T'* (write tree for RHS that ends in T_1', T_2').
  - *By instantiating the quatified variables (sigmas and b's) and using the other conditions from the property* $P(T_2)$ *must hold by induction hypothesis. Therefore there exists a derivation tree* T'' *... By choosing* $T_2' = T''$ *the claim is satisfied and case is concluded.*

Axiomatic Semantics / Hoarre Triple Proofs

- Total Correctness : don't forget if we have `while i < k`, then for the conditions inside the while loop we need : {Z=i - k} , Z a fresh variable, and at the end : {i - k < Z}
- **Semantic Equivalence** = s_1, s_2 are semantically equivalent $\equiv$ $\forall P, Q. \vdash \{P\}s_1\{Q\} \Leftrightarrow \{P\}s_2\{Q\}$

Safety & Liveness Properties

- A formula is a liveness property if we can extend any finite string such that it does not violate the property
- A formula is a safety property if whenever it is violated for some instance then it must be violated in a finite prefix of that instance