Tuan

**Question 9.1:**

> data Nat = Zero | Succ Nat

> deriving (Eq, Ord, Show)

> int :: Nat -> Int

> int Zero = 0

> int (Succ x) = 1 + int x

> nat :: Int -> Nat

> nat 0 = Zero

> nat n = Succ (nat (n - 1))

> add :: Nat -> Nat -> Nat

> add n Zero = n

> add n (Succ m) = Succ (add n m)

> mul :: Nat -> Nat -> Nat

> mul n Zero = Zero

> mul n (Succ m) = n `add` (mul n m)

> pow :: Nat -> Nat -> Nat

> pow n Zero = Succ Zero

> pow n (Succ m) = n `mul` (pow n m)

> tet :: Nat -> Nat -> Nat

> tet n Zero = Zero

> tet n (Succ Zero) = n

> tet n (Succ m) = (tet n m) `pow` n

**Question 9.2:**

The fold function will iterate through the whole Nat list, and the last empty list will be replaced by Nil.

> foldNat :: (a -> a) -> a -> Nat -> a

> foldNat cons nil Zero = nil

> foldNat cons nil (Succ n) = cons (foldNat cons nil n)

> unfoldNat :: (a -> Bool) -> (Nat -> Nat) -> (a -> a) -> a -> Nat

> unfoldNat null head tail x = if (not . null) x then head (unfoldNat null head tail (tail x)) else Zero

> intFold :: Nat -> Int

> intFold = foldNat (1+) 0

> natFold :: Int -> Nat

> natFold = unfoldNat (== 0) Succ (subtract 1)

> add' :: Nat -> Nat -> Nat

> add' n m = foldNat Succ n m

> mul' :: Nat -> Nat -> Nat

> mul' n m = foldNat (`add` n) Zero m

> pow' :: Nat -> Nat -> Nat

> pow' n m = foldNat (`mul` n) (Succ Zero) m

Fix tet'

> tet' :: Nat -> Nat -> Nat

> tet' n m = foldNat (n `pow`) (Succ Zero) m

**Question 10.1:**

Case 1: *xs = []*

*fold c n ([] ++ ys)*

*= {definition of (++)}*

*fold c n ys*

*fold c (fold c n ys) []*

*= {definition of fold}*

*fold c n ys*

So LHS = RHS

Case 2: *xs = undefined*

*fold c n (undefined ++ ys)*

*= {strictness of (++)}*

*fold c n undefined*

*= {strictness of fold}*

*undefined*

*fold c (fold c n ys) undefined*

*= {strictness of fold}*

*Undefined*

So LHS = RHS

Case 3: *x : xs*

*fold c n (x : xs ++ ys)*

*= {definition of ++}*

*fold c n (x : (xs ++ ys))*

*= {definition of fold}*

*c x (fold c n (xs ++ ys))*

*= {induction hypothesis}*

*c x (fold c (fold c n ys) xs)*

*= {definition of fold}*

*fold c (fold c n ys) (x : xs)*, which is the same as the RHS.

The function is chain complete, and by proving for these three cases, it follows that

*fold c n (xs ++ ys) = fold c (fold c n ys) xs*

**Question 10.2:**

There are three requirements to use fold fusion:

1. (*++ bs*) is strict
2. (*++ bs*) [] = bs
3. (*++ bs*) . *(:) = (:) . (++ bs)*

*(++ bs) . fold (:) [] = fold (:) bs*

*Foldr c n (xs ++ ys) = foldr c n (foldr (:) ys xs)*

To use fold fusion we need three requirements:

1. *foldr c n is strict,* which is true
2. *foldr c n ys = foldr c n ys*
3. *c x (foldr c n y) = foldr c n (x : y) = foldr c n ((:) x y),* so h x (f y) = f (g x y)

So *foldr c n (foldr (:) ys xs) = foldr c (foldr c n ys) xs*

**Question 10.3:**

1. *filter p undefined = undefined*
2. *filter p [] = []*
3. *filter p . (x :) = h x . filter p*, where *h x ys = if p x then x : ys else ys*

So *filter p = foldr h []*, where *h x ys = if p x then x : ys else ys*

*filter p (xs ++ ys)*

*= fold h [] (xs ++ ys)*

*= fold h (fold h ys) xs*

*= fold (:) (fold h ys) (fold h xs)*

*= fold h xs ++ fold h ys*

*= filter p xs ++ filter h ys*

**Question 10.4:**

> data Liste a = Snoc (Liste a) a | Lin

> deriving Show

> cat :: Liste a -> Liste a -> Liste a

> cat xs Lin = xs

> cat xs (Snoc ys a) = Snoc (cat xs ys) a

> folde :: (a -> b -> b) -> b -> Liste a -> b

> folde cons nil Lin = nil

> folde cons nil (Snoc n a) = cons a (folde cons nil n)

> cat' :: Liste a -> Liste a -> Liste a

> cat' xs = folde func xs

> where func a b = Snoc b a

> list :: Liste a -> [a]

> list = folde f []

> where f x = (++[x])

> liste :: [a] -> Liste a

> liste = foldr f Lin

> where f x Lin = Snoc Lin x

> f x (Snoc n a) = Snoc (f x n) a

*liste* returns bottom when applied to an infinite list. The end is not well defined in the infinite object of type Liste a.

> revfolde :: (b -> a -> b) -> b -> Liste a -> b

> revfolde cons nil Lin = nil

> revfolde cons nil (Snoc n a) = revfolde cons (cons nil a) n

> tailfold :: (b -> a -> b) -> b -> [a] -> b

> tailfold c n [] = n

> tailfold c n (x : xs) = tailfold c (c n x) xs

> list' :: Liste a -> [a]

> list' = revfolde func []

> where func a b = b : a

> liste' :: [a] -> Liste a

> liste' = foldl func Lin

> where func a b = Snoc a b

**Question 10.5:**

> unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]

> unfold n h t x

> | n x = []

> | otherwise = h x : unfold n h t (t x)

> unfolde :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> Liste b

> unfolde n h t x

> | n x = Lin

> | otherwise = Snoc (unfolde n h t (t x)) (h x)

> list'' :: Liste a -> [a]

> list'' = unfold n h t

> where n Lin = True

> n \_ = False

> h (Snoc Lin a) = a

> h (Snoc n a) = h n

> t (Snoc Lin a) = Lin

> t (Snoc n a) = Snoc (t n) a

> liste'' :: [a] -> Liste a

> liste'' = unfolde n last init

> where n [] = True

> n \_ = False