```haskell
-- Model solution for Lab09
--
-- Copyright [2000..2004] Manuel M T Chakravarty

module Lab09
where

mulNats   :: Int -> Int
mulNats 0  = 1
mulNats n  = n * mulNats (n - 1)

natList   :: Int -> [Int]
natList 0  = []
natList n  = n : natList (n - 1)

prodList        :: [Int] -> Int
prodList []      = 1
prodList (x:xs)  = x * prodList xs

{- -----------------------------

Property: P(n) ===   mulNats n = prodList (natList n)

PROOF:

Induction over n.

Base: P(0) ===   mulNats 0 = prodList (natList 0)

 Left side:

    mulNats 0
  = {mulNats.1}
    1

 Right side:

    prodList (natList 0)
  = {natList.1}
    prodList []
  = {prodList.1}
    1

Induction: P(n + 1) ===   mulNats (n + 1) = prodList (natList (n + 1))

 Left side:

    mulNats (n + 1)
  = {n + 1 > 0 and mulNats.2}
    (n + 1) * mulNats (n + 1 - 1)
  = {Arithmetic}
    (n + 1) * mulNats n
  = {Induction hypothesis}
    (n + 1) * prodList (natList n)

 Right side:

    prodList (natList (n + 1))
  = {natList.2}
    prodList ((n + 1) : (natList (n + 1 - 1))
  = {Arithmetic}
    prodList ((n + 1) : (natList n))
  = {prodList}.2
    (n + 1) * prodList (natList n)

QED

-}
```

```
concat           :: [[a]] -> [a]
concat []         = []
concat (xs:xss)   = xs ++ concat xss

(++)          :: [a] -> [a] -> [a]
[]      ++ ys   = ys
(x:xs) ++ ys   = x : (xs ++ ys)</pre>

{- ----------------------------

Property: P(xss) ===   concat (xss ++ yss) = concat xss ++ concat yss

PROOF:

Structural induction over xss.

Base: P([]) ===   concat ([] ++ yss) = concat [] ++ concat yss

 Left side:

    concat ([] ++ yss)
  = {++.1}
    concat yss

 Right side:

    concat [] ++ concat yss
  = {concat.1}
    concat yss

Base: P(xs:xss) ===   concat ((xs:xss) ++ yss) = concat (xs:xss) ++ concat yss

 Left side:

    concat ((xs:xss) ++ yss)
  = {++.2}
    concat (xs:(xss ++ yss))
  = {concat.2}
    xs ++ concat (xss ++ yss)
  = {Induction hypothesis}
    xs ++ concat xss ++ concat yss

 Right side:

    concat (xs:xss) ++ concat yss
  = {concat.2}
    xs ++ concat xss ++ concat yss

QED
-}

reverse        :: [a] -> [a]
reverse []      = []
reverse (x:xs) = reverse xs ++ [x]

{- ----------------------------

Property: P(xs) ===   reverse (reverse xs) = xs

PROOF:

Structural induction over xs.

Base: P([]) ===   reverse (reverse []) = []

   reverse (reverse [])
 = {reverse.1}
   reverse []
 = {reverse.1}
```

```
   []

Induction: P(x:xs) ===   reverse (reverse (x:xs)) = (x:xs)

 Left side:

    reverse (reverse (x:xs))
  = {reverse.2}
    reverse (reverse xs ++ [x])

 Right side:

    x:xs
  = {Induction hypothesis}
    x : reverse (reverse xs)

This leaves us with a requirement to prove that

  reverse (reverse xs ++ [x]) = x : reverse (reverse xs)

We generalise this to

  Q(ys) ===   reverse (ys ++ [x]) = x : reverse ys

Base: Q([]) ===   reverse ([] ++ [x]) = x : reverse []

 Left side:

    reverse ([] ++ [x])
  = {++.1}
    reverse [x]
  = {reverse.2}
    reverse [] ++ [x]
  = {reverse.1}
    [] ++ [x]
  = {++.1}
    [x]

 Right side:

    x : reverse []
  = {reverse.1}
    x:[]
  =
    [x]

Induction: Q(y:ys) ===   reverse ((y:ys) ++ [x]) = x : reverse (y:ys)

 Left side:

    reverse ((y:ys) ++ [x])
  = {++.2}
    reverse (y:(ys ++ [x]))
  = {reverse.2}
    reverse (ys ++ [x]) ++ [y]
  = {Induction hypothesis}
    (x : reverse ys) ++ [y]
  = {++.2}
    x : (reverse ys ++ [y])

 Right side:

    x : reverse (y:ys)
  = {reverse.2}
    x : (reverse ys ++ [y])

QED
-}
```