```
{- Computed datatypes -}

{- From these basic datatypes... -}

data One               = unit
data (*)  (A,B :: Set) = pair (a :: A) (b :: B)

{- ... construct the vectors of a given length -}

data Nat = zero | suc (n :: Nat)

Vec :: Nat -> Set -> Set
Vec n X = case n of
  (zero)  -> One
  (suc n) -> X * Vec n X

{- safe destructors for nonempty vectors -}

vHead (X :: Set) :: (n :: Nat)-> Vec (suc n) X -> X
vHead n xs = case xs of
                (pair a b)-> a

vTail (X :: Set) :: (n :: Nat)-> Vec (suc n) X -> Vec n X
vTail n xs = case xs of
                (pair a b)-> b

{- useful vector programming operators -}

vec (X :: Set) :: (n :: Nat)-> X -> Vec n X
vec n x = case n of
             (zero)-> unit
             (suc n')-> pair x (vec n' x )

vapp (S,T :: Set) :: (n :: Nat) -> Vec n (S -> T) -> Vec n S -> Vec n T
vapp n fs ss = case n of
  (zero)-> unit
  (suc n')-> case fs of
    (pair f fs')-> case ss of
      (pair s ss')-> pair (f s ) (vapp n' fs' ss' )

{- mapping and zipping come from these -}

vMap (S,T :: Set) :: (n :: Nat)-> (S -> T) -> Vec n S -> Vec n T
vMap n f ss = vapp n (vec n f) ss

{- transposition gets the type it deserves -}

transpose (X :: Set) :: (m,n :: Nat)-> Vec m (Vec n X) -> Vec n (Vec m X)
transpose m n xss = case m of
  (zero)->   vec n unit
  (suc m')-> case xss of
    (pair xs xss')-> vapp n (vapp n (vec n pair) xs) (transpose m' n xss' )

{- Resist the temptation to mention Idioms. -}

{- Sets of a given finite size may be computed as follows... -}

data Zero              =
data (+)  (A,B :: Set) = inl (a :: A) | inr (b :: B)

Fin :: Nat -> Set
Fin n = case n of
  (zero)-> Zero
  (suc n')-> One + Fin n'

{- We can use these sets to index vectors safely. -}

vProj (X :: Set) :: (n :: Nat)-> Vec n X -> Fin n -> X
```

```
vProj n xs i = case n of
  (zero)-> case i of { }
  (suc n')-> case i of
    (inl u)-> case xs of
      (pair x xs')-> x
    (inr i')-> case xs of
      (pair x xs')-> vProj n' xs' i'

{- We can also tabulate a function as a vector. Resist
   the temptation to mention logarithms. -}

vTab (X :: Set) :: (n :: Nat)-> (Fin n -> X) -> Vec n X
vTab n f = case n of
  (zero)-> unit
  (suc n')-> pair (f (inl unit ))
                  (vTab n' (\x -> f (inr x) ))

{- Question to ponder in your own time:
   if we use functional vectors, what are vec and vapp? -}

{- Inductive datatypes of the unfocused variety -}

{- Every constructor must target the whole family, rather
   than focusing on specific indices. -}

data Tm (n :: Nat) :: Set
  = var (i :: Fin n)
  | app (f,s :: Tm n)
  | lda (t :: Tm (suc n))

{- Renamings -}

Ren :: Nat -> Nat -> Set
Ren m n = Vec m (Fin n)

{- identity and composition -}

idR :: (n :: Nat)-> n `Ren` n
idR n = vTab n (\i -> i)

coR :: (l,m,n :: Nat)-> m `Ren` n -> l `Ren` m -> l `Ren` n
coR l m n m2n l2m = vMap l (vProj m m2n) l2m

{- what theorems should we prove? -}

{- the lifting functor for Ren -}

liftR :: (m,n :: Nat)-> m `Ren` n -> suc m `Ren` suc n
liftR m n m2n = pair (inl unit ) (vMap m inr m2n )

{- what theorems should we prove? -}

{- the functor from Ren to Tm-arrows -}

rename (m,n :: Nat) :: (m `Ren` n) -> Tm m -> Tm n
rename m2n t = case t of
  (var i)-> var (vProj m m2n i)
  (app f s)-> app (rename m2n f) (rename m2n s)
  (lda t')-> lda (rename (liftR m n m2n) t')

{- Substitutions -}

Sub :: Nat -> Nat -> Set
Sub m n = Vec m (Tm n)

{- identity; composition must wait; why? -}

idS :: (n :: Nat)-> n `Sub` n
idS n = vTab n var
```

```
{- functor from renamings to substitutions -}

Ren2Sub :: (m,n :: Nat)-> m `Ren` n -> m `Sub` n
Ren2Sub m n m2n = vMap m (\(x :: Fin n) -> var x) m2n

{- lifting functor for substitution -}

liftS :: (m,n :: Nat)-> m `Sub` n -> suc m `Sub` suc n
liftS m n m2n = pair (var (inl unit ))
                     (vMap m (rename (vMap n inr (idR n) )) m2n )

{- functor from Sub to Tm-arrows -}

subst (m,n :: Nat) :: m `Sub` n -> Tm m -> Tm n
subst m2n t = case t of
  (var i)-> vProj m m2n i
  (app f s)-> app (subst m2n f) (subst m2n s)
  (lda t')-> lda (subst (liftS m n m2n ) t' )

{- and now we can define composition -}

coS :: (l,m,n :: Nat)-> m `Sub` n -> l `Sub` m -> l `Sub` n
coS l m n m2n l2m = vMap l (subst m2n) l2m
```