

module HuttonChap16 where

```
open import Haskell.Prelude
open import Haskell.Law.Equality using (sym; begin_; _≡⟨_⟩_; step-≡; _■; cong)
open import Haskell.Law.Num.Def using (+-assoc; +-comm)
open import Haskell.Law.Num.Int using (iLawfulNumInt)
```

INDUCTION ON NUMBERS

Proving the first fact about replicate:

```
replicate : {a : Set} → Nat → a → List a
replicate zero _ = []
replicate (suc n) x = x :: replicate n x
```

```
len-repl : {A : Set} → (n : Nat) → (x : A) → lengthNat (replicate n x) ≡ n
len-repl zero x = refl
len-repl (suc n) x =
  begin
    lengthNat (replicate (suc n) x)
  ≡⟨⟩ -- Apply replicate
    lengthNat (x :: replicate n x)
  ≡⟨⟩ -- Apply lengthNat
    suc (lengthNat (replicate n x))
  ≡⟨ cong suc (len-repl n x) ⟩
    suc n
  ■
```

Some facts about append:

```
++-[] : {a : Set} → (xs : List a) → xs ++ [] ≡ xs
++-[] [] = begin ([] ++ []) ≡⟨⟩ [] ■
++-[] (x :: xs) =
  begin
    (x :: xs) ++ []
  ≡⟨⟩ -- Apply ++
    x :: (xs ++ [])
  ≡⟨ cong (x ::_) (++-[] xs) ⟩
    x :: xs
  ■
```

```
++-assoc : {a : Set} → (xs ys zs : List a)
→ (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc [] ys zs =
  begin
    ([] ++ ys) ++ zs
  ≡⟨⟩ -- Apply ++
    ys ++ zs
  ≡⟨⟩ -- Unapply ++
    [] ++ (ys ++ zs)
  ■
```

```

++-assoc (x :: xs) ys zs =
  begin
    ((x :: xs) ++ ys) ++ zs
  ≡⟨ -- Apply ++
    (x :: (xs ++ ys)) ++ zs
  ≡⟨ -- Apply ++
    x :: ((xs ++ ys) ++ zs)
  ≡⟨ cong (x ::_) (++-assoc xs ys zs) ⟩
    x :: (xs ++ (ys ++ zs))
  ≡⟨ -- Unapply ++
    (x :: xs) ++ (ys ++ zs)
  ■

```

Hutton's example of elimination of append from flattening a tree:

```

data Tree (a : Set) : Set where
  Leaf : a → Tree a
  Node : Tree a → Tree a → Tree a
{-# COMPILE AGDA2HS Tree #-}

flatten : {a : Set} → Tree a → List a
flatten (Leaf x) = x :: []
flatten (Node tl tr) = flatten tl ++ flatten tr
{-# COMPILE AGDA2HS flatten #-}

flatten' : {a : Set} → Tree a → List a → List a
flatten' (Leaf x) xs = x :: xs
flatten' (Node tl tr) xs = flatten' tl (flatten' tr xs)
{-# COMPILE AGDA2HS flatten' #-}

flatten'-flatten : {a : Set} → (t : Tree a) → (xs : List a)
  → flatten' t xs ≡ flatten t ++ xs
flatten'-flatten (Leaf x) xs = refl
flatten'-flatten (Node tl tr) xs =
  begin
    flatten' (Node tl tr) xs
  ≡⟨ -- Apply flatten'
    flatten' tl (flatten' tr xs)
  ≡⟨ cong (flatten' tl) (flatten'-flatten tr xs) ⟩
    flatten' tl (flatten tr ++ xs)
  ≡⟨ flatten'-flatten tl (flatten tr ++ xs) ⟩
    flatten tl ++ (flatten tr ++ xs)
  ≡⟨ sym (++-assoc (flatten tl) (flatten tr) xs) ⟩
    (flatten tl ++ flatten tr) ++ xs
  ≡⟨ -- Unapply flatten
    flatten (Node tl tr) ++ xs
  ■

```

```

flatten'≡-flatten : {a : Set} → (t : Tree a)
  → flatten' t [] ≡ flatten t
flatten'≡-flatten (Leaf x) = refl
flatten'≡-flatten (Node tl tr) =
  begin
    flatten' (Node tl tr) []
  ≡⟨ ⟩ -- Apply flatten'
    flatten' tl (flatten' tr [])
  ≡⟨ cong (flatten' tl) (flatten'-flatten tr []) ⟩ -- Apply the above equality
    flatten' tl (flatten tr ++ [])
  ≡⟨ flatten'-flatten tl (flatten tr ++ []) ⟩ -- Apply it again
    flatten tl ++ (flatten tr ++ [])
  ≡⟨ cong (flatten tl ++-) (++-[]) (flatten tr) ⟩ -- Remove trailing []
    flatten tl ++ flatten tr
  ≡⟨ ⟩ -- Unapply flatten
    flatten (Node tl tr)
  ■

```

COMPILER CORRECTNESS

```

data Expr : Set where
  Val : Int → Expr
  Add : Expr → Expr → Expr
{-# COMPILE AGDA2HS Expr #-}

eval : Expr → Int
eval (Val n) = n
eval (Add x y) = eval x + eval y
{-# COMPILE AGDA2HS eval #-}

Stack = List Int
{-# COMPILE AGDA2HS Stack #-}

data Op : Set where
  PUSH : Int → Op
  ADD : Op
{-# COMPILE AGDA2HS Op #-}

Code = List Op
{-# COMPILE AGDA2HS Code #-}

exec : Code → Stack → Stack
exec [] s = s
exec (PUSH n :: c) s = exec c (n :: s)
exec (ADD :: c) (m :: n :: s) = exec c (n + m :: s)
exec (ADD :: c) _ = []
{-# COMPILE AGDA2HS exec #-}

comp : Expr → Code
comp (Val n) = PUSH n :: []
comp (Add el er) = comp el ++ comp er ++ ADD :: []
{-# COMPILE AGDA2HS comp #-}

module CompilerCorrectness where
  comp' : Expr → Code → Code
  comp' (Val n) c = PUSH n :: c
  comp' (Add x y) c = comp' x $ comp' y (ADD :: c)
  {-# COMPILE AGDA2HS comp' #-}

```

```

comp'-exec-eval : (e : Expr) → (c : Code) → (s : Stack)
  → exec (comp' e c) s ≡ exec c (eval e :: s)
comp'-exec-eval (Val n) c s =
  begin
    exec (comp' (Val n) c) s
  ≡⟨⟩ -- Apply comp'
    exec (PUSH n :: c) s
  ≡⟨⟩ -- Apply exec
    exec c (n :: s)
  ≡⟨⟩ -- Unapply eval
    exec c (eval (Val n) :: s)
  ■

comp'-exec-eval (Add x y) c s =
  begin
    exec (comp' (Add x y) c) s
  ≡⟨⟩ -- Apply comp'
    exec (comp' x $ comp' y $ ADD :: c) s
  ≡⟨ comp'-exec-eval x (comp' y $ ADD :: c) s ⟩ -- Induction
    exec (comp' y $ ADD :: c) (eval x :: s)
  ≡⟨ comp'-exec-eval y (ADD :: c) (eval x :: s) ⟩ -- Induction Again
    exec (ADD :: c) (eval y :: eval x :: s)
  ≡⟨⟩ -- Apply exec
    exec c ((eval x) + (eval y) :: s)
  ≡⟨⟩ -- Unapply eval
    exec c (eval (Add x y) :: s)
  ■

compile : Expr → Code
compile e = comp' e []
{-# COMPILE AGDA2HS compile #-}

compile-exec-eval : (e : Expr) → exec (compile e) [] ≡ eval e :: []
compile-exec-eval e =
  begin
    exec (compile e) []
  ≡⟨⟩ -- Apply compile
    exec (comp' e []) []
  ≡⟨ comp'-exec-eval e [] [] ⟩
    exec [] (eval e :: [])
  ≡⟨⟩ -- Apply exec
    eval e :: []
  ■

```

EXERCISE 1. Show that $\text{add } n \text{ (Suc } m) = \text{Suc } (\text{add } n \text{ } m)$ by induction on n

```

+-suc : (n m : Nat) → n + (suc m) ≡ suc (n + m)
+-suc zero m = refl
+-suc (suc n) m =
  begin
    (suc n) + (suc m)
  ≡⟨⟩ -- Apply +
    suc (n + suc m)
  ≡⟨ cong suc (+-suc n m) ⟩
    suc (suc (n + m))
  ≡⟨⟩ -- Unapply +
    suc (suc n + m)
  ■

```

EXERCISE 2. Using this property, together with $\text{add } n \text{ zero} = n$, show that addition is commutative, $\text{add } n \ m = \text{add } m \ n$, by induction on n .

```

+-zero : (n : Nat) → n + zero ≡ n
+-zero zero = refl
+-zero (suc n) =
  begin
    suc n + zero
  ≡⟨⟩ -- Apply +
    suc (n + zero)
  ≡⟨ cong suc (+-zero n) ⟩
    suc n
  ■

```

```

+-commut : (n m : Nat) → n + m ≡ m + n
+-commut zero m =
  begin
    zero + m
  ≡⟨⟩ -- Apply +
    m
  ≡⟨ sym (+-zero m) ⟩
    m + zero
  ■

```

```

+-commut (suc n) m =
  begin
    suc n + m
  ≡⟨⟩ -- Apply +
    suc (n + m)
  ≡⟨ cong suc (+-commut n m) ⟩
    suc (m + n)
  ≡⟨ sym (+-suc m n) ⟩
    m + suc n
  ■

```

EXERCISE 3. Complete the proof of the correctness of replicate by showing that it produces a list with identical elements, $\text{all } (== \ x) \ (\text{replicate } n \ x)$, by induction on $n \geq 0$. Hint: show that the property is always True.

```

open import Haskell.Law.Eq.Def using (IsLawfulEq; eqReflexivity)
all-repl : {a} iEq : Eq a → {a} IsLawfulEq a → (n : Nat) → (x : a)
  → all (_== x) (replicate n x) ≡ True
all-repl zero x = refl
all-repl (suc n) x =
  begin
    all (_== x) (replicate (suc n) x)
  ≡⟨⟩ -- Apply replicate
    all (_== x) (x :: replicate n x)
  ≡⟨⟩ -- Apply all
    (x == x) && (all (_== x) (replicate n x))
  ≡⟨ cong ((x == x) &&_) (all-repl n x) ⟩ -- Induction
    (x == x) && True
  ≡⟨ cong (_&& True) (eqReflexivity x) ⟩ -- Reflexivity x == x
    True
  ■

```

EXERCISE 4. This is $++$ -[] and $++$ -assoc above.

EXERCISE 5. Using the above definition for $++$, together with the definitions for take and drop show that $\text{take } n \text{ } xs \text{ } ++ \text{ drop } n \text{ } xs = xs$, by simultaneous induction on the integer n and the list xs . Hint: there are three cases, one for each pattern of arguments in the definitions of take and drop .

```
take-drop-nat : {a : Set} → (n : Nat) → (xs : List a)
  → takeNat n xs ++ dropNat n xs ≡ xs
take-drop-nat n [] = refl
take-drop-nat zero (x :: xs) =
  begin
    takeNat zero (x :: xs) ++ dropNat zero (x :: xs)
  ≡⟨⟩ -- Apply takeNat and dropNat
    [] ++ x :: xs
  ≡⟨⟩
    x :: xs
  ■

take-drop-nat (suc n) (x :: xs) =
  begin
    takeNat (suc n) (x :: xs) ++ dropNat (suc n) (x :: xs)
  ≡⟨⟩ -- Apply takeNat and dropNat and ++
    x :: takeNat n xs ++ dropNat n xs
  ≡⟨ cong (x ::_) (take-drop-nat n xs) ⟩
    x :: xs
  ■
```

```
take-drop : {a : Set} → (n : Int) → (iNN : IsNonNegativeInt n) →
  → (xs : List a) → take n xs ++ drop n xs ≡ xs
take-drop n xs =
  begin
    take n xs ++ drop n xs
  ≡⟨⟩ -- Apply take and drop
    takeNat (intToNat n) xs ++ dropNat (intToNat n) xs
  ≡⟨ take-drop-nat (intToNat n) xs ⟩
    xs
  ■
```

EXERCISE 6. Given the `Tree` definition above, show that the number of leaves in such a tree is always one greater than the number of nodes, by induction on trees. Hint: start by defining functions that count the number of leaves and nodes in a tree.

```
nLeaves : {a : Set} → Tree a → Int
nLeaves (Leaf x) = 1
nLeaves (Node tl tr) = nLeaves tl + nLeaves tr
{-# COMPILE AGDA2HS nLeaves #-}

nNodes : {a : Set} → Tree a → Int
nNodes (Leaf x) = 0
nNodes (Node tl tr) = 1 + nNodes tl + nNodes tr
{-# COMPILE AGDA2HS nNodes #-}
```

```

leaves-nodes : {a : Set} → (t : Tree a) → nLeaves t ≡ 1 + nNodes t
leaves-nodes (Leaf x) = refl
leaves-nodes (Node tl tr) =
  begin
    nLeaves (Node tl tr)
  ≡⟨⟩
    nLeaves tl + nLeaves tr
  ≡⟨ cong (λ_+ (nLeaves tr)) (leaves-nodes tl) ⟩
    1 + nNodes tl + nLeaves tr
  ≡⟨ cong ((1 + nNodes tl) +_) (leaves-nodes tr) ⟩
    1 + nNodes tl + (1 + nNodes tr)
  ≡⟨ +-assoc 1 (nNodes tl) (1 + nNodes tr) ⟩
    1 + (nNodes tl + (1 + nNodes tr))
  ≡⟨ cong (1 +_) (sym (+-assoc (nNodes tl) 1 (nNodes tr))) ⟩
    1 + (nNodes tl + 1 + nNodes tr)
  ≡⟨ cong (1 +_) (cong (λ_+ nNodes tr) (+-comm (nNodes tl) 1)) ⟩
    1 + (1 + nNodes tl + nNodes tr)
  ≡⟨⟩
    1 + nNodes (Node tl tr)
  ■

```

EXERCISE 7. Verify the functor laws for the Maybe type. Hint: the proofs proceed by case analysis, and do not require the use of induction.

```

module FunctorLawsMaybe where
identity : {a : Set} → (m : Maybe a) → (fmap id) m ≡ id m
identity Nothing =
  begin
    fmap id Nothing
  ≡⟨⟩ -- Apply fmap
    Nothing
  ≡⟨⟩ -- Unapply id
    id Nothing
  ■
identity (Just x) =
  begin
    fmap id (Just x)
  ≡⟨⟩ -- Apply fmap
    Just (id x)
  ≡⟨⟩ -- Apply id
    Just x
  ≡⟨⟩ -- Unapply id
    id (Just x)
  ■

```

```

composition : {a b c : Set}
  → (m : Maybe a) → (f : a → b) → (g : b → c)
  → fmap (g ∘ f) m ≡ (fmap g ∘ fmap f) m
composition Nothing f g =
  begin
    fmap (g ∘ f) Nothing
  ≡⟨⟩ -- Apply fmap
    Nothing
  ≡⟨⟩ -- Unapply fmap
    fmap g Nothing
  ≡⟨⟩ -- Unapply fmap
    fmap g (fmap f Nothing)
  ≡⟨⟩ -- Unapply ∘
    (fmap g ∘ fmap f) Nothing
  ■

```

```

composition (Just x) f g =
  begin
    fmap (g ∘ f) (Just x)
  ≡⟨⟩ -- Apply fmap
    Just ((g ∘ f) x)
  ≡⟨⟩ -- Apply ∘
    Just (g (f x))
  ≡⟨⟩ -- Unapply fmap
    fmap g (Just (f x))
  ≡⟨⟩ -- Unapply fmap
    fmap g (fmap f (Just x))
  ≡⟨⟩ -- Unapply ∘
    (fmap g ∘ fmap f) (Just x)
  ■

```

```

module LawfulFunctorMaybe where
open import Haskell.Law.Functor.Def
  using (IsLawfulFunctor; identity; composition)
instance
  isLawful : IsLawfulFunctor Maybe
  identity ∯ isLawful ∅ = FunctorLawsMaybe.identity
  composition ∯ isLawful ∅ = FunctorLawsMaybe.composition

```

EXERCISE 8. Given the instance declaration below, verify the functor laws for the `Tree` type, by induction on trees.

```

open import Haskell.Prim.Functor using (DefaultFunctor)
treeMap : {a b : Set} → (a → b) → (Tree a) → (Tree b)
treeMap f (Leaf x) = Leaf (f x)
treeMap f (Node tl tr) = Node (treeMap f tl) (treeMap f tr)
{-# COMPILE AGDA2HS treeMap #-}

dft : DefaultFunctor Tree
dft = record { fmap = treeMap }
instance
  iFunctorTree : Functor Tree
  iFunctorTree = record { DefaultFunctor dft }
  {-# COMPILE AGDA2HS iFunctorTree #-}

```



```

module FunctorLawsTree where
  identity : (t : Tree a) → (fmap id) t ≡ id t
  identity (Leaf x) = refl
  identity (Node tl tr) =
    begin
      fmap id (Node tl tr)
    ≡⟨⟩ -- Apply fmap
      Node (fmap id tl) (fmap id tr)
    ≡⟨ cong (λ x → Node x (fmap id tr)) (identity tl) ⟩
      Node (id tl) (fmap id tr)
    ≡⟨ cong (Node (id tl)) (identity tr) ⟩
      Node (id tl) (id tr)
    ≡⟨⟩ -- Apply and unapply id
      id (Node tl tr)
    ■

  composition : (t : Tree a) → (f : a → b) → (g : b → c)
    → fmap (g ∘ f) t ≡ (fmap g ∘ fmap f) t
  composition (Leaf x) f g = refl
  composition (Node tl tr) f g =
    begin
      fmap (g ∘ f) (Node tl tr)
    ≡⟨⟩ -- Apply fmap
      Node (fmap (g ∘ f) tl) (fmap (g ∘ f) tr)
    ≡⟨ cong (λ x → Node x (fmap (g ∘ f) tr)) (composition tl f g) ⟩ -- Induction
      Node ((fmap g ∘ fmap f) tl) (fmap (g ∘ f) tr)
    ≡⟨ cong (Node ((fmap g ∘ fmap f) tl)) (composition tr f g) ⟩ -- Induction
      Node ((fmap g ∘ fmap f) tl) ((fmap g ∘ fmap f) tr)
    ≡⟨⟩ -- Unapply fmap
      fmap g (Node (fmap f tl) (fmap f tr))
    ≡⟨⟩ -- Unapply fmap
      fmap g (fmap f (Node tl tr))
    ≡⟨⟩ -- Unapply ∘
      (fmap g ∘ fmap f) (Node tl tr)
    ■

module LawfulFunctorTree where
  open import Haskell.Law.Functor.Def
  using (IsLawfulFunctor; identity; composition)
  instance
    isLawful : IsLawfulFunctor Tree
    identity ⌈ isLawful ⌋ = FunctorLawsTree.identity
    composition ⌈ isLawful ⌋ = FunctorLawsTree.composition

```

EXERCISE 9. Verify the applicative laws for the Maybe type.

```

module ApplicativeLawsMaybe where
  identity : {a : Set} → (m : Maybe a) → (pure id <*> m) ≡ m
  identity Nothing =
    begin
      pure id <*> Nothing
    ≡⟨⟩ -- Apply pure and <*>
      Nothing
    ■

```

```
identity (Just x) =
  begin
    pure id <*> Just x
  ≡⟨⟩ -- Apply pure
    Just id <*> Just x
  ≡⟨⟩ -- Apply <*>
    Just (id x)
  ≡⟨⟩ -- Apply id
    Just x
```

■

```
composition : {a b c : Set}
  → (x : Maybe (b → c)) → (y : Maybe (a → b)) → (z : Maybe a)
  → (pure _◦_ <*> x <*> y <*> z) ≡ (x <*> (y <*> z))
```

```
composition Nothing y z =
  begin
    pure _◦_ <*> Nothing <*> y <*> z
  ≡⟨⟩ -- Apply pure and <*>
    Nothing <*> y <*> z
  ≡⟨⟩ -- Apply the rest of the <*>
    Nothing
  ≡⟨⟩ -- Unapply <*> on the right
    Nothing <*> (y <*> z)
```

■

```
composition (Just x) Nothing z =
  begin
    pure _◦_ <*> Just x <*> Nothing <*> z
  ≡⟨⟩ -- Apply pure and <*>
    Just (x ◦_) <*> Nothing <*> z
  ≡⟨⟩ -- Apply <*>
    Nothing <*> z
  ≡⟨⟩ -- Apply <*>
    Nothing
  ≡⟨⟩ -- Unapply <*>
    Nothing <*> z
  ≡⟨⟩ -- Unapply <*>
    Just x <*> (Nothing <*> z)
```

■

```
composition (Just x) (Just y) Nothing =
  refl -- Same kind of proof as above.
```

```
composition (Just x) (Just y) (Just z) =
  begin
    pure _◦_ <*> Just x <*> Just y <*> Just z
  ≡⟨⟩ -- Apply pure and <*>
    Just (x ◦_) <*> Just y <*> Just z
  ≡⟨⟩ -- Apply <*>
    Just (x ◦ y) <*> Just z
  ≡⟨⟩ -- Apply <*>
    Just ((x ◦ y) z)
  ≡⟨⟩ -- Apply ◦
    Just (x (y z))
  ≡⟨⟩ -- Unapply <*>
    Just x <*> Just (y z)
  ≡⟨⟩ -- Unapply <*>
    Just x <*> (Just y <*> Just z)
```

■

```

homomorphism : {a b : Set} → (f : a → b) (x : a)
→ ((pure f) <*> (pure x)) ≡ (pure (f x))
homomorphism f x =
begin
  pure f <*> pure x
≡⟨⟩ -- Apply pure
  Just f <*> Just x
≡⟨⟩ -- Apply <*>
  Just (f x)
≡⟨⟩ -- Unapply pure
  pure (f x)
■

```

```

interchange : {a b : Set} → (x : Maybe (a → b)) (y : a)
→ (x <*> (pure y)) ≡ (pure (λ f → f y) <*> x)
interchange Nothing y =
begin
  Nothing <*> pure y
≡⟨⟩ -- Apply <*>
  Nothing
≡⟨⟩ -- Unapply <*>
  pure (λ $ y) <*> Nothing
■

```

```

interchange (Just x) y =
begin
  (Just x) <*> pure y
≡⟨⟩ -- Apply <*>
  Just (x y)
≡⟨⟩ -- Unapply $
  Just ((λ $ y) x)
≡⟨⟩ -- Unapply <*>
  pure (λ $ y) <*> Just x
■

```

```

module LawfulApplicativeMaybe where
open import Haskell.Law.Applicative.Def
using (IsLawfulApplicative; identity; composition;
homomorphism; interchange; functor)
instance
  isLawful : IsLawfulApplicative Maybe
  identity ∷ isLawful ∷ = ApplicativeLawsMaybe.identity
  composition ∷ isLawful ∷ = ApplicativeLawsMaybe.composition
  homomorphism ∷ isLawful ∷ = ApplicativeLawsMaybe.homomorphism
  interchange ∷ isLawful ∷ = ApplicativeLawsMaybe.interchange
  functor ∷ isLawful ∷ f Nothing = refl -- These are by definition.
  functor ∷ isLawful ∷ f (Just x) = refl

```

EXERCISE 10. Verify the monad laws for the list type. Hint: the proofs can be completed using simple properties of list comprehensions.

```

module MonadLawsList where
  leftIdentity : {a : Set} → (x : a) (f : a → List b)
    → ((return x) >=> f) ≡ f x
  leftIdentity x f =
    begin
      (return x) >=> f
    ≡⟨⟩ -- Apply return
      (x :: []) >=> f
    ≡⟨⟩ -- Apply >=>
      f x ++ []
    ≡⟨ +-[] (f x) ⟩
      f x
    ■

  fmap2bind : {a b : Set} → (f : a → b) → (xs : List a)
    → fmap f xs ≡ (xs >=> (return ∘ f))
  fmap2bind f [] = refl
  fmap2bind f (x :: xs) =
    begin
      fmap f (x :: xs)
    ≡⟨⟩ -- Apply fmap
      (f x :: []) ++ fmap f xs
    ≡⟨ cong ((f x :: []) ++_) (fmap2bind f xs) ⟩ -- Induction
      (f x :: []) ++ (xs >=> (return ∘ f))
    ≡⟨⟩ -- Unapply return
      (return (f x)) ++ (xs >=> (return ∘ f))
    ≡⟨⟩ -- Unapply ∘
      (return ∘ f) x ++ (xs >=> (return ∘ f))
    ≡⟨⟩ -- Unapply >=>
      (x :: xs) >=> (return ∘ f)
    ■

  import Haskell.Law.Functor as Functor
  rightIdentity : {a : Set} → (xs : List a) → (xs >=> return) ≡ xs
  rightIdentity xs =
    begin
      xs >=> return
    ≡⟨⟩ -- Unapply id
      xs >=> (return ∘ id)
    ≡⟨ sym (fmap2bind id xs) ⟩ -- Apply fmap2bind
      map id xs
    ≡⟨ Functor.identity xs ⟩ -- Apply Functor Law: identity
      xs
    ■

```

```

++-distrib : {a : Set} → (xs ys : List a) → (f : a → List b)
  → ((xs >=> f) ++ (ys >=> f)) ≡ ((xs ++ ys) >=> f)
++-distrib [] ys f = refl
++-distrib (x :: xs) ys f =
  begin
    ((x :: xs) >=> f) ++ (ys >=> f)
  ≡⟨⟩ -- Apply >=>
    (f x ++ (xs >=> f)) ++ (ys >=> f)
  ≡⟨ +-assoc (f x) (xs >=> f) (ys >=> f) ⟩ -- Move parens
    f x ++ (xs >=> f) ++ (ys >=> f)
  ≡⟨ cong (f x ++_) (++-distrib xs ys f) ⟩ -- Induction
    f x ++ ((xs ++ ys) >=> f)
  ≡⟨⟩ -- Unapply >=>
    (x :: (xs ++ ys)) >=> f
  ≡⟨⟩ -- Unapply ++
    ((x :: xs) ++ ys) >=> f
  ■

```

```

associativity : {a b c : Set}
  → (xs : List a) → (f : a → List b) → (g : b → List c)
  → (xs >=> λ x → f x >=> g) ≡ ((xs >=> f) >=> g)
associativity [] f g = refl
associativity (x :: xs) f g =
  begin
    (x :: xs) >=> (λ x → f x >=> g)
  ≡⟨⟩ -- Apply >=>
    (f x >=> g) ++ (xs >=> λ x → f x >=> g)
  ≡⟨ cong ((f x >=> g) ++_) (associativity xs f g) ⟩ -- Induction
    (f x >=> g) ++ ((xs >=> f) >=> g)
  ≡⟨ +-distrib (f x) (xs >=> f) g ⟩ -- Move Parens
    (f x ++ (xs >=> f)) >=> g
  ≡⟨⟩ -- Unapply inner >=>
    ((x :: xs) >=> f) >=> g
  ■

```

```

sequence2bind : {a b : Set}
  → (fs : List (a → b)) → (xs : List a)
  → (fs <*> xs) ≡ (fs >=> λ f → (xs >=> (return ∘ f)))
sequence2bind [] xs = refl
sequence2bind (f :: fs) xs =
  begin
    f :: fs <*> xs
  ≡⟨⟩ -- Apply <*>
    fmap f xs ++ (fs <*> xs)
  ≡⟨ cong (_++ (fs <*> xs)) (fmap2bind f xs) ⟩ -- Apply fmap2bind
    xs >=> (return ∘ f) ++ (fs <*> xs)
  ≡⟨ cong (xs >=> (return ∘ f) ++_) (sequence2bind fs xs) ⟩ -- Induction
    xs >=> (return ∘ f)
    ++ (fs >=> λ f → (xs >=> (return ∘ f)))
  ≡⟨⟩ -- Unapply λ
    ((λ f → (xs >=> (return ∘ f))) f)
    ++ (fs >=> λ f → (xs >=> (return ∘ f)))
  ≡⟨⟩ -- Unapply >=>
    (f :: fs) >=> (λ f → (xs >=> (return ∘ f)))
  ■

```

```

rSequence2rBind : {a b : Set} → (xs : List a) → (ys : List b)
  → (xs *> ys) ≡ (xs >> ys)
rSequence2rBind [] ys = refl
rSequence2rBind (x :: xs) ys =
  begin
    (x :: xs) *> ys
  ≡⟨⟩ -- Apply *>
    (const id x) :: (fmap (const id) xs) <*> ys
  ≡⟨⟩ -- Apply <*>
    fmap (const id x) ys ++ (fmap (const id) xs <*> ys)
  ≡⟨⟩ -- Apply const
    fmap id ys ++ (fmap (const id) xs <*> ys)
  -- Functor Law: identity
  ≡⟨ cong (λ_++ (fmap (const id) xs <*> ys)) (Functor.identity ys) ⟩
    ys ++ (fmap (const id) xs <*> ys)
  ≡⟨⟩ -- Unapply *>
    ys ++ (xs *> ys)
  ≡⟨ cong (ys ++_) (rSequence2rBind xs ys) ⟩ -- Induction
    ys ++ (xs >> ys)
  ≡⟨⟩ -- Apply >>
    ys ++ (xs >>= const ys)
  ≡⟨⟩ -- Unapply const
    (const ys) x ++ (xs >>= const ys)
  ≡⟨⟩ -- Unapply >>=
    (x :: xs) >>= const ys
  ≡⟨⟩ -- Unapply >>
    (x :: xs) >> ys
  ■

```

```

module LawfulMonadList where
open import Haskell.Law.Applicative.List
open import Haskell.Law.Monad.Def
  using (IsLawfulMonad; leftIdentity; rightIdentity;
    associativity; pureIsReturn; sequence2bind; fmap2bind;
    rSequence2rBind)

instance
  isLawful : IsLawfulMonad List
  leftIdentity  ⚡ isLawful ⚡ = MonadLawsList.leftIdentity
  rightIdentity ⚡ isLawful ⚡ = MonadLawsList.rightIdentity
  associativity ⚡ isLawful ⚡ = MonadLawsList.associativity
  pureIsReturn  ⚡ isLawful ⚡ _ = refl -- By definition
  sequence2bind ⚡ isLawful ⚡ = MonadLawsList.sequence2bind
  fmap2bind     ⚡ isLawful ⚡ = MonadLawsList.fmap2bind
  rSequence2rBind ⚡ isLawful ⚡ = MonadLawsList.rSequence2rBind

```

EXERCISE 11. Given the equation $\text{comp}' \ e \ c = \text{comp} \ e \ ++ \ c$, show how to construct the recursive definition for comp' , by induction on e .

```

module Exercise11 where
postulate
  comp' : Expr → Code → Code
  comp'-comp : (e : Expr) → (c : Code) → comp' e c ≡ comp e ++ c

```

```

comp'-val : (n : Int) → (c : Code) → comp' (Val n) c ≡ (PUSH n :: c)
comp'-val n c =
  begin
    comp' (Val n) c
  ≡⟨ comp'-comp (Val n) c ⟩
    comp (Val n) ++ c
  ≡⟨ ⟩ -- Apply comp
    PUSH n :: [] ++ c
  ≡⟨ ⟩ -- Apply ++
    PUSH n :: c
  ■

```

```

comp'-add : (el er : Expr) → (c : Code)
→ comp' (Add el er) c ≡ comp' el (comp' er (ADD :: c))
comp'-add el er c =
  begin
    comp' (Add el er) c
  ≡⟨ comp'-comp (Add el er) c ⟩ -- Postulate
    comp (Add el er) ++ c
  ≡⟨ ⟩ -- Apply comp
    (comp el ++ comp er ++ ADD :: []) ++ c
  ≡⟨ ++-assoc (comp el) (comp er ++ ADD :: []) c ⟩ -- Move parens
    comp el ++ (comp er ++ ADD :: []) ++ c
  ≡⟨ cong (comp el ++_) (++-assoc (comp er) (ADD :: []) c) ⟩ -- Move parens
    comp el ++ (comp er ++ ADD :: [] ++ c)
  ≡⟨ ⟩ -- Apply ++
    comp el ++ comp er ++ ADD :: c
  ≡⟨ cong (comp el ++_) (sym (comp'-comp er (ADD :: c))) ⟩ -- Induction
    comp el ++ comp' er (ADD :: c)
  ≡⟨ sym (comp'-comp el (comp' er (ADD :: c))) ⟩ -- Induction
    comp' el (comp' er (ADD :: c))
  ■

```