```
module HuttonChap16 where

open import Haskell.Prelude
open import Haskell.Law.Equality using (sym; begin_; _≡⟨⟩_; step-≡; _∎; cong)

++-[] : {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 []) ⟩ -- Inner induction
    flatten' tₗ (flatten tᵣ ++ [])
  ≡⟨ flatten'-flatten tₗ (flatten tᵣ ++ []) ⟩ -- Induction again
    flatten tₗ ++ (flatten tᵣ ++ [])
  ≡⟨ cong (flatten tₗ ++_) (++-[] (flatten tᵣ)) ⟩ -- Remove trailing []
    flatten tₗ ++ flatten tᵣ
  ≡⟨⟩ -- Unapply flatten
    flatten (Node tₗ tᵣ)
  ∎
```