```
module HuttonChap17 where
open import Haskell.Prelude
open import Haskell.Law.Equality using (sym; begin_; _≡⟨⟩_; step-≡; _∎; cong)
data Expr : Set where
    Val : Int → Expr
    Add : Expr → Expr → Expr
{-# COMPILE AGDA2HS Expr #-}

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

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

push : Int → Stack → Stack
push n s = n ∷ s
{-# COMPILE AGDA2HS push #-}

add : Stack → Stack
add [] = []
add (x ∷ []) = []
add (x ∷ y ∷ s) = y + x ∷ s

postulate
  eval' : Expr → Stack → Stack
  eval'-eval : (e : Expr) → (s : Stack) → eval' e s ≡ eval e ∷ s

eval'-val : (n : Int) → (s : Stack)
  → eval' (Val n) s ≡ push n s
eval'-val n s =
  begin
    eval' (Val n) s
  ≡⟨ eval'-eval (Val n) s ⟩ -- Specification
    eval (Val n) ∷ s
  ≡⟨⟩ -- Apply eval
    n ∷ s
  ≡⟨⟩ -- Unapply push
    push n s
  ∎
```

```
eval'-add : (x y : Expr) → (s : Stack)
  → eval' (Add x y) s ≡ add (eval' y (eval' x s))
eval'-add x y s =
  begin
    eval' (Add x y) s
  ≡⟨ eval'-eval (Add x y) s ⟩ -- Specification
    eval (Add x y) ∷ s
  ≡⟨⟩ -- Apply eval
    eval x + eval y ∷ s
  ≡⟨⟩ -- Unapply add
    add (eval y ∷ eval x ∷ s)
  ≡⟨ cong (λ x → add (eval y ∷ x)) (sym (eval'-eval x s)) ⟩ -- Induction
    add (eval y ∷ eval' x s)
  ≡⟨ cong add (sym (eval'-eval y (eval' x s))) ⟩
    add (eval' y (eval' x s))
  ∎
```