```
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 #-}

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
{-# COMPILE AGDA2HS add #-}

module Naïve where
  eval : Expr → Int
  eval (Val n) = n
  eval (Add eₗ eᵣ) = eval eₗ + eval eᵣ

module DefineEval' where
  open Naïve

  eval'-val : (eval' : Expr → Stack → Stack)
    → (eval'-eval : (e : Expr) → (s : Stack) → eval' e s ≡ eval e ∷ s)
    → (n : Int) → (s : Stack) → eval' (Val n) s ≡ push n s
  eval'-val eval' eval'-eval 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 : (eval' : Expr → Stack → Stack)
    → (eval'-eval : (e : Expr) → (s : Stack) → eval' e s ≡ eval e ∷ s)
    → (x y : Expr) → (s : Stack)
    → eval' (Add x y) s ≡ add (eval' y (eval' x s))
  eval'-add eval' eval'-eval 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 s → add (eval y ∷ s)) (sym (eval'-eval x s)) ⟩ -- Induction
      add (eval y ∷ eval' x s)
    ≡⟨ cong add (sym (eval'-eval y (eval' x s))) ⟩ -- Induction
      add (eval' y (eval' x s))
    ∎

eval' : Expr → Stack → Stack
eval' (Val n) s = push n s
eval' (Add eₗ eᵣ) s = add (eval' eᵣ (eval' eₗ s))
{-# COMPILE AGDA2HS eval' #-}

eval'≡eval : (e : Expr) → (s : Stack) → eval' e s ≡ Naïve.eval e ∷ s
eval'≡eval (Val n) s = refl
eval'≡eval (Add x y) s =
  begin
    eval' (Add x y) s
  ≡⟨⟩ -- Apply eval'
    add (eval' y (eval' x s))
  ≡⟨ cong (λ s → add (eval' y s)) (eval'≡eval x s) ⟩ -- Induction
    add (eval' y (Naïve.eval x ∷ s))
  ≡⟨ cong add (eval'≡eval y (Naïve.eval x ∷ s)) ⟩ -- Induction
    add (Naïve.eval y ∷ Naïve.eval x ∷ s)
  ≡⟨⟩ -- Unapply add and Naïve.eval
    Naïve.eval (Add x y) ∷ s
  ∎
eval≡eval' : (e : Expr) → (s : Stack) → Naïve.eval e ∷ s ≡ eval' e s
eval≡eval' e s = sym $ eval'≡eval e s

open import Haskell.Prim using (NonEmpty; itsNonEmpty)
open import Haskell.Law.Equality using (subst)

eval'-nonempty : (e : Expr) → NonEmpty (eval' e [])
eval'-nonempty e = subst NonEmpty (eval≡eval' e []) itsNonEmpty

eval : Expr → Int
eval e = head (eval' e [])
  where instance
    ne : NonEmpty (eval' e [])
    ne = eval'-nonempty e
{-# COMPILE AGDA2HS eval #-}
```

```
Cont = Stack → Stack
{-# COMPILE AGDA2HS Cont #-}

module DefineEval'' where
  postulate
    eval'' : Expr → Cont → Cont
    eval''-eval' : (e : Expr) → (c : Cont) → (s : Stack)
      → eval'' e c s ≡ c (eval' e s)

  eval''-val : (n : Int) → (c : Cont) → (s : Stack)
    → eval'' (Val n) c s ≡ c (push n s)
  eval''-val n c s =
    begin
      eval'' (Val n) c s
    ≡⟨ eval''-eval' (Val n) c s ⟩ -- Postulate
      c (eval' (Val n) s)
    ≡⟨⟩ -- Apply eval'
      c (push n s)
    ∎
  eval''-add : (x y : Expr) → (c : Cont) → (s : Stack)
    → eval'' (Add x y) c s ≡ eval'' x (eval'' y (c ∘ add)) s
  eval''-add x y c s =
    begin
      eval'' (Add x y) c s
    ≡⟨ eval''-eval' (Add x y) c s ⟩
      c (eval' (Add x y) s)
    ≡⟨⟩ -- Apply eval'
      c (add (eval' y (eval' x s)))
    ≡⟨⟩ -- Unapply ∘
      (c ∘ add) (eval' y (eval' x s))
    ≡⟨ sym (eval''-eval' y (c ∘ add) (eval' x s)) ⟩ -- Induction y
      eval'' y (c ∘ add) (eval' x s)
    ≡⟨ sym (eval''-eval' x (eval'' y (c ∘ add)) s) ⟩ -- Induction x
      eval'' x (eval'' y (c ∘ add)) s
    ∎

eval'' : Expr → Cont → Cont
eval'' (Val n) c = c ∘ push n
eval'' (Add x y) c = eval'' x (eval'' y (c ∘ add))
{-# COMPILE AGDA2HS eval'' #-}

eval'¹ : Expr → Cont
eval'¹ e = eval'' e id
{-# COMPILE AGDA2HS eval'¹ #-}
```