

```

module HuttonChap17 where
open import Haskell.Prelude
open import Haskell.Law.Equality using (sym; begin_;  $\equiv$ <>_; step- $\equiv$ ;  $\dashv$ █; cong)

```

## 1. SYNTAX AND SEMANTICS

```

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 #-}

```

## 2. ADDING A STACK

```

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

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

add : Stack → Stack
add [] = []
add (x :: []) = [x]
add (x :: y :: s) = y + x :: s
{-# COMPILE AGDA2HS add #-}

module DefineEval' where
  postulate
    eval' : Expr → Stack → Stack
    eval'≡eval : (e : Expr) → (s : Stack) → eval' e s  $\equiv$  eval e :: s

  eval'-val : (n : Int) → (s : Stack) → eval' (Val n) s  $\equiv$  push n s
  eval'-val n s =
    begin
      eval' (Val n) s
     $\equiv$ < eval'≡eval (Val n) s > -- Specification
      eval (Val n) :: s
     $\equiv$ <> -- Apply eval
      n :: s
     $\equiv$ <> -- Unapply push
      push n s
    █

  eval'-add : (x y : Expr) → (s : Stack)
    → eval' (Add x y) s  $\equiv$  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 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 ≡ 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 (eval x :: s))
  ≡⟨ cong add (eval'≡eval y (eval x :: s)) ⟩ -- Induction
    add (eval y :: eval x :: s)
  ≡⟨ ⟩ -- Unapply add and eval
    eval (Add x y) :: s
  ■

```

```

eval≡eval' : (e : Expr) → (s : Stack) → eval e :: s ≡ eval' e s
eval≡eval' e s = sym $ eval'≡eval e s

```

Since  $\text{eval}' e s$  is the same as  $\text{eval } e :: s$ , this is evidence that  $\text{eval}' e s$  is a non-empty list:

```

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

```

```

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

```

Since we know that  $\text{eval}' e []$  is always non-empty, it is safe to apply head to it and get an equivalent definition of eval:

```

eval1 : ∀ Expr → Int
eval1 e = head (eval' e [])
{-# COMPILE AGDA2HS eval1 #-}

```

### 3. ADDING A CONTINUATION

Cont = Stack  $\rightarrow$  Stack

{-# COMPILE AGDA2HS Cont #-}

module DefineEval'' where

postulate

eval'' : Expr  $\rightarrow$  Cont  $\rightarrow$  Cont

eval'' $\equiv$ eval' : (e : Expr)  $\rightarrow$  (c : Cont)  $\rightarrow$  (s : Stack)

$\rightarrow$  eval'' e c s  $\equiv$  c (eval' e s)

eval''-val : (n : Int)  $\rightarrow$  (c : Cont)  $\rightarrow$  (s : Stack)

$\rightarrow$  eval'' (Val n) c s  $\equiv$  c (push n s)

eval''-val n c s =

begin

eval'' (Val n) c s

$\equiv$  ( eval'' $\equiv$ eval' (Val n) c s ) -- Postulate

c (eval' (Val n) s)

$\equiv$  ( ) -- Apply eval'

c (push n s)

$\equiv$  ( ) -- Unapply  $\circ$

(c  $\circ$  push n) s

■

eval''-add : (x y : Expr)  $\rightarrow$  (c : Cont)  $\rightarrow$  (s : Stack)

$\rightarrow$  eval'' (Add x y) c s  $\equiv$  eval'' x (eval'' y (c  $\circ$  add)) s

eval''-add x y c s =

begin

eval'' (Add x y) c s

$\equiv$  ( eval'' $\equiv$ eval' (Add x y) c s )

c (eval' (Add x y) s)

$\equiv$  ( ) -- Apply eval'

c (add (eval' y (eval' x s)))

$\equiv$  ( ) -- Unapply  $\circ$

(c  $\circ$  add) (eval' y (eval' x s))

$\equiv$  ( sym (eval'' $\equiv$ eval' y (c  $\circ$  add) (eval' x s)) ) -- Induction y

eval'' y (c  $\circ$  add) (eval' x s)

$\equiv$  ( sym (eval'' $\equiv$ eval' x (eval'' y (c  $\circ$  add)) s) ) -- Induction x

eval'' x (eval'' y (c  $\circ$  add)) s

■

eval'' : Expr  $\rightarrow$  Cont  $\rightarrow$  Cont

eval'' (Val n) c = c  $\circ$  push n

eval'' (Add x y) c = eval'' x (eval'' y (c  $\circ$  add))

{-# COMPILE AGDA2HS eval'' #-}

eval'' $\equiv$ eval' : (e : Expr)  $\rightarrow$  (c : Cont)  $\rightarrow$  (s : Stack)

$\rightarrow$  eval'' e c s  $\equiv$  c (eval' e s)

eval'' $\equiv$ eval' (Val x) c s = refl

```

eval''≡eval' (Add x y) c s =
  begin
    eval'' (Add x y) c s
  ≡⟨⟩ -- Apply eval''
    eval'' x (eval'' y (c ∘ add)) s
  ≡⟨ eval''≡eval' x (eval'' y (c ∘ add)) s ⟩ -- Induction
    eval'' y (c ∘ add) (eval' x s)
  ≡⟨ eval''≡eval' y (c ∘ add) (eval' x s) ⟩ -- Induction
    (c ∘ add) (eval' y (eval' x s))
  ≡⟨⟩ -- Apply add
    c (eval' (Add x y) s)
■

```

Thus `eval'` is simply redefined as follows:

```

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

```

## 4. DEFUNCTIONALISING

```

haltC : Cont
haltC = id
{-# COMPILE AGDA2HS haltC #-}

pushC : Int → Cont → Cont
pushC n c = c ∘ push n
{-# COMPILE AGDA2HS pushC #-}

addC : Cont → Cont
addC c = c ∘ add
{-# COMPILE AGDA2HS addC #-}

data Code : Set where
  HALT : Code
  PUSH : Int → Code → Code
  ADD : Code → Code
{-# COMPILE AGDA2HS Code deriving Show #-}

exec : Code → Cont
exec HALT = haltC
exec (PUSH n c) = pushC n (exec c)
exec (ADD c) = addC (exec c)
{-# COMPILE AGDA2HS exec #-}

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 : Expr → Code
comp e = comp' e HALT
{-# COMPILE AGDA2HS comp #-}

```

```

exec-comp'≡eval'' : (e : Expr) → (c : Code)
  → exec (comp' e c) ≡ eval'' e (exec c)
exec-comp'≡eval'' (Val n) c = refl
exec-comp'≡eval'' (Add x y) c =
  begin
    exec (comp' (Add x y) c)
  ≡⟨⟩ -- Apply comp'
    exec (comp' x (comp' y (ADD c)))
  ≡⟨ exec-comp'≡eval'' x (comp' y (ADD c)) ⟩ -- Induction
    eval'' x (exec (comp' y (ADD c)))
  ≡⟨ cong (eval'' x) (exec-comp'≡eval'' y (ADD c)) ⟩ -- Induction
    eval'' x (eval'' y (exec (ADD c)))
  ≡⟨⟩ -- Apply exec
    eval'' x (eval'' y (addC (exec c)))
  ≡⟨⟩ -- Unapply eval''
    eval'' (Add x y) (exec c)
  ■

exec-comp≡eval' : (e : Expr) → (s : Stack) → exec (comp e) s ≡ eval' e s
exec-comp≡eval' e s =
  begin
    exec (comp e) s
  ≡⟨⟩ -- Apply comp
    exec (comp' e HALT) s
  ≡⟨ cong (λ$ s) (exec-comp'≡eval'' e HALT) ⟩
    eval'' e (exec HALT) s
  ≡⟨⟩ -- Apply exec
    eval'' e id s
  ≡⟨ eval''≡eval' e id s ⟩
    id (eval' e s)
  ≡⟨⟩ -- Apply id
    eval' e s
  ■

```

Alternatively, explicitly with lists:

```

data Op : Set where
  PUSHOP : Int → Op
  ADDOP : Op
{-# COMPILER AGDA2HS Op #-}

Prog = List Op
{-# COMPILER AGDA2HS Prog #-}

execute : Prog → Cont
execute [] = haltC
execute (PUSHOP n :: os) = pushC n (execute os)
execute (ADDOP :: os) = addC (execute os)
{-# COMPILER AGDA2HS execute #-}

compile' : Expr → Prog → Prog
compile' (Val n) p = PUSHOP n :: p
compile' (Add x y) p = compile' x (compile' y (ADDOP :: p))
{-# COMPILER AGDA2HS compile' #-}

compile : Expr → Prog
compile e = compile' e []
{-# COMPILER AGDA2HS compile #-}

```

```

execute-compile'≡eval'' : (e : Expr) → (p : Prog)
  → execute (compile' e p) ≡ eval'' e (execute p)
execute-compile'≡eval'' (Val n) p = refl
execute-compile'≡eval'' (Add x y) p =
  begin
    execute (compile' (Add x y) p)
  ≡⟨⟩ -- Apply compile'
    execute (compile' x (compile' y (ADDOP :: p)))
  ≡⟨ execute-compile'≡eval'' x (compile' y (ADDOP :: p)) ⟩ -- Induction
    eval'' x (execute (compile' y (ADDOP :: p)))
  ≡⟨ cong (eval'' x) (execute-compile'≡eval'' y (ADDOP :: p)) ⟩
    eval'' x (eval'' y (execute $ ADDOP :: p))
  ≡⟨⟩ -- Apply execute
    eval'' x (eval'' y (addC (execute p)))
  ≡⟨⟩ -- apply addC
    eval'' x (eval'' y ((execute p) ∘ add))
  ≡⟨⟩ -- Unapply eval''
    eval'' (Add x y) (execute p)
  ■

```

```

execute-compile≡eval' : (e : Expr) → (s : Stack)
  → execute (compile e) s ≡ eval' e s
execute-compile≡eval' e s =
  begin
    execute (compile e) s
  ≡⟨⟩ -- Apply compile
    execute (compile' e []) s
  ≡⟨ cong (_$ s) (execute-compile'≡eval'' e []) ⟩
    eval'' e (execute []) s
  ≡⟨⟩ -- Apply execute
    eval'' e haltC s
  ≡⟨ eval''≡eval' e haltC s ⟩
    haltC (eval' e s)
  ≡⟨⟩ -- Apply haltC ≡ id
    eval' e s
  ■

```

```

execute-compile≡eval : (e : Expr) → (s : Stack)
  → execute (compile e) s ≡ eval e :: s
execute-compile≡eval e s =
  begin
    execute (compile e) s
  ≡⟨ execute-compile≡eval' e s ⟩
    eval' e s
  ≡⟨ eval'≡eval e s ⟩
    eval e :: s
  ■

```

## 5. COMBINING THE STEPS