

# Chapter 16

`data Nat = Zero | Succ Nat deriving Show`

```
add :: Nat -> Nat -> Nat
add Zero m = m
add (Succ n) m = Succ $ add n m
```

```
reverse' :: [a] -> [a] -> [a]
reverse' [] ys = ys
reverse' (x : xs) ys = reverse' xs (x : ys)
```

`data Tree = Leaf Int | Node Tree Tree`

```
flatten :: Tree -> [Int]
flatten (Leaf n) = [n]
flatten (Node l r) = flatten l ++ flatten r
```

We define a relation

$$\text{flatten}' \ t \ ns = \text{flatten } t \ ++ \ ns$$

and use this to derive `flatten'` as follows. Base case:

$$\begin{aligned} \text{flatten}' \ (\text{Leaf } n) \ ns &= \text{flatten } (\text{Leaf } n) \ ++ \ ns \\ &= [n] \ ++ \ ns \\ &= n :: ns \end{aligned}$$

Induction:

$$\begin{aligned} \text{flatten}' \ (\text{Node } l \ r) \ ns &= \text{flatten } l \ ++ \ \text{flatten } r \ ++ \ ns \\ &= \text{flatten } l \ ++ \ \text{flatten}' \ r \ ns \\ &= \text{flatten}' \ l \ \$ \ \text{flatten}' \ r \ ns \end{aligned}$$

So we define:

```
flatten' :: Tree -> [Int] -> [Int]
flatten' (Leaf n) ns = n : ns
flatten' (Node l r) ns = flatten' l $ flatten' r ns

flatten2 :: Tree -> [Int]
flatten2 t = flatten' t []
```

## Compiler correctness

`data Expr = Val Int | Add Expr Expr deriving Show`

```
eval :: Expr -> Int
eval (Val n) = n
eval (Add x y) = eval x + eval y
```

```

type Stack = [Int]
type Code = [Op]
data Op = PUSH Int | ADD deriving Show

exec :: Code -> Stack -> Stack
exec [] s = s
exec (PUSH x : ops) s = exec ops (x : s)
exec (ADD : ops) (x : y : s) = exec ops (y + x : s)

comp :: Expr -> Code
comp (Val x) = [PUSH x]
comp (Add e1 er) = comp e1 ++ comp er ++ [ADD]

e :: Expr
e = Add (Add (Val 2) (Val 3)) (Val 4)

```

Compiler law version 1:

$$\text{exec (comp e) []} = [\text{eval e}]$$

Compiler law version 2:

$$\text{exec (comp e) s} = \text{eval e} : s$$

This is proved in the book but the proof needs the distributivity property proved below.

The distributivity property is that `exec` distributes over lists, that executing two pieces of code appended together is the same as executing the two pieces of code in sequence:

$$\text{exec (c ++ d) s} = \text{exec d \$ exec c s}$$

Inductive Case:

$$\begin{aligned}
 \text{exec ((PUSH n : c) ++ d) s} &= \text{exec (PUSH n : c ++ d) s} && \text{Apply ++} \\
 &= \text{exec (c ++ d) (n : s)} && \text{Apply exec} \\
 &= \text{exec d \$ exec c (n : s)} && \text{Apply induction} \\
 &= \text{exec d \$ exec (PUSH n : c) s} && \text{Unapply exec}
 \end{aligned}$$

Other inductive case:

$$\begin{aligned}
 \text{exec ((ADD : c) ++ d) s} &= \text{exec (Add : c ++ d) (x : y : s')} && \text{Apply ++} \\
 &= \text{exec (c ++ d) (y + x : s')} && \text{Apply exec} \\
 &= \text{exec d \$ exec c (y + x : s')} && \text{Apply induction} \\
 &= \text{exec d \$ exec (ADD : c) s} && \text{Unapply exec}
 \end{aligned}$$

Making append vanish with `comp`: a new `comp` that satisfies:

$$\text{comp' e c} = \text{comp e ++ c}$$

This will allow us to get rid of

$\text{comp } (\text{Add } e_1 \text{ er}) = \text{comp } e_1 ++ \text{comp } er ++ [\text{ADD}]$

to replace it with

$\text{comp}' (\text{Add } e_1 \text{ er}) \text{ c} = \text{comp}' e_1 \$ \text{comp}' er \$ \text{ADD} : \text{c}$

In fact that's the definition:

```
comp' :: Expr -> Code -> Code
comp' (Val x) c = PUSH x : c
comp' (Add e1 er) c = comp' e1 $ comp' er $ ADD : c
```

With this new  $\text{comp}'$  the validity with respect to  $\text{eval}$  is:

$$\text{exec } (\text{comp}' e \text{ c}) \text{ s} = \text{exec } c \$ \text{eval } e : \text{s}$$

The advantage of this new definition is we do not need the distributivity lemma, but can prove this directly:

Base case:

$$\begin{aligned} \text{exec } (\text{comp}' (\text{Val } x) \text{ c}) \text{ s} &= \text{exec } (\text{PUSH } x : \text{c}) \text{ s} \\ &= \text{exec } c \$ x : \text{s} \\ &= \text{exec } c \$ \text{eval } (\text{Val } x) : \text{s} \end{aligned}$$

Inductive case:

$$\begin{aligned} \text{exec } (\text{comp}' (\text{Add } e_1 \text{ er}) \text{ c}) \text{ s} \\ &= \text{exec } (\text{comp}' e_1 \$ \text{comp}' er \$ \text{ADD} : \text{c}) \text{ s} \\ &= \text{exec } (\text{comp}' er \$ \text{ADD} : \text{c}) \$ \text{eval } e_1 : \text{s} \\ &= \text{exec } (\text{ADD} : \text{c}) \$ \text{eval } er : \text{eval } e_1 : \text{s} \\ &= \text{exec } c \$ \text{eval } e_1 + \text{eval } er : \text{s} \\ &= \text{exec } c \$ \text{eval } (\text{Add } e_1 \text{ er}) : \text{s} \end{aligned}$$

## Exercises

EXERCISE 1: Show that  $\text{add } n \$ \text{Succ } m = \text{Succ } \$ \text{add } n \text{ m}$  by induction on  $n$ .