Laws and Equations and Coq

Dave Laing

March 26, 2013

Outline

Outline

Outline

Outline

Laws

Laws

- Help when reasoning about code
- Particularly code that makes use of typeclasses
- There's only so much that a typeclass can enforce
- For everything else there is
  - Equational reasoning
  - Theorem provers
  - Angry letters to the authors

Monoids are great!

- Taking the tree out of tree shaped computations
  - Summarize with monoids and Data.Foldable
  - Fun times with finger trees

Monoids

A monoid is a combination of

- a type `A`

and a function

- $\oplus$ :: $\mathtt{A} \rightarrow \mathtt{A} \rightarrow \mathtt{A}$

that satisfies certain laws.

Monoid Laws

There must exist and element `e :: A` such that for all `a :: A`

- `e` $\oplus$ `a = a` (Left identity)
- `a` $\oplus$ `e = a` (Right identity)

For all `a, b, c :: A`

- `a` $\oplus$ `(b` $\oplus$ `c) = (a` $\oplus$ `b)` $\oplus$ `c` (Associativity)

Monoid typeclass and List instance

```
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a

instance Monoid [a] where
    mempty = []
    mappend = (++)
```

(Remembering that `(++)` has type `[a] -> [a] -> [a]`)

Functors are great!

- Apply a function to everything in a structure
  - You can double the values in a tree
- Preserve the shape of the structure

- Compose functions and apply once
  - You can double and then add one to the values in a tree
- Saves multiple traversals

Aside: Type constructors

Suppose `F` is a type constructor (like Maybe or List)

- `F` isn't a type
- `F Int`, `F String` are types
- we'll use `F a`, `F b` as types for abstraction

Functors

A functor is a combination of

- a type constructor `F`

and

- a function `fmap :: (a → b) → F a → F b`

that satisfies certain laws.

Functor Laws

- `fmap id = id` (Identity)
- `fmap g (fmap f xs) = fmap (g` $\circ$ `f) xs` (Composition)

Functor typeclass and List instance

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map
```

(Remember that `map` has type `(a -> b) -> [a] -> [b]`)

Induction

Natural numbers

- `O :: Nat`
  - this is zero
- `S :: Nat -> Nat`
  - this is the successor function
  - it returns one more than its input
- `Nat = O | S(x)`
  - `x` $\in$ `Nat`
  - so the definition of `Nat` is recursive
- Ascii time!

```
  S         S              S                  ...
 / \       / \            / \
O   S(O)    S(S(O))    S(S(S(O)))  ...
```

Addition

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
S(S(0)) + S(S(S(0)))
```

2 + 3

Addition

```
(+) :: Nat -> Nat -> Nat
0     + n = n
S(m) + n = S(m + n)
```

`S(S(0)) + S(S(S(0)))` 2 + 3

Addition

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
S(S(0)) + S(S(S(0)))                    2 + 3
```

Addition

```
(+) :: Nat -> Nat -> Nat
0      + n = n
S(m) + n = S(m + n)
```

```
S(S(0)) + S(S(S(0)))                    2 + 3
```

Addition

```
(+) :: Nat -> Nat -> Nat
O     + n = n
S(m) + n = S(m + n)
```

```
S(S(O)  + S(S(S(O))))
```

1+ (1 + 3)

Addition

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
S(S(0)  + S(S(S(0))))                  1+ (1 + 3)
```

Addition

```
(+) :: Nat -> Nat -> Nat
0     + n = n
S(m) + n = S(m + n)
```

```
S(S(0)  + S(S(S(0))))                    1+ (1 + 3)
```

Addition

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
S(S(0)  + S(S(S(0))))                    1+ (1 + 3)
```

Addition

```
(+) :: Nat -> Nat -> Nat
O     + n = n
S(m) + n = S(m + n)
```

```
S(S(O   + S(S(S(O)))))          1+ 1+ (0 + 3)
```

Addition

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
S(S(0   + S(S(S(0)))))          1+ 1+ (0 + 3)
```

Addition

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
S(S(0   + S(S(S(0)))))          1+ 1+ (0 + 3)
```

Addition

```
(+) :: Nat -> Nat -> Nat
0     + n = n
S(m) + n = S(m + n)
```

```
S(S      (S(S(S(0)))))                    1+ 1+ 3
```

Addition

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
S(S    (S(S(S(0)))))                    5
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0        + 0 = ?
```

Right identity - the long way

Goal: $\forall$ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = ?
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = ?
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = ?
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = ?
```

Right identity - the long way

Goal: $\forall$ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = ?
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = ?
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)


0       + 0 = 0
S(0)    + 0 = ?
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0        + 0 = 0
S(0)     + 0 = S(0 + 0)
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0 + 0)
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0        + 0 = 0
S(0)     + 0 = S(0 + 0)
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0      + 0 = 0
S(0)   + 0 = S(0 + 0)
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0)
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0)
S(S(0)) + 0 = ?
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0)
S(S(0)) + 0 = ?
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0)
S(S(0)) + 0 = ?
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0)
S(S(0)) + 0 = ?
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0)
S(S(0)) + 0 = S(S(0) + 0)
```

Right identity - the long way

Goal: $\forall$ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0)
S(S(0)) + 0 = S(S(0) + 0)
```

Right identity - the long way

Goal: $\forall$ `x :: Nat`

```
x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0)
S(S(0)) + 0 = S(S(0) + 0)
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
  x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0)
S(S(0)) + 0 = S(S(0) + 0)
```

Right identity - the long way

Goal: $\forall$ `x :: Nat`

```
x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0)
S(S(0)) + 0 = S(S(0))
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0)
S(S(0)) + 0 = S(S(0))
```

Right identity - the long way

Goal: ∀ `x :: Nat`

```
x + 0 = x
```

```
(+) :: Nat -> Nat -> Nat
0    + n = n
S(m) + n = S(m + n)
```

```
0       + 0 = 0
S(0)    + 0 = S(0)
S(S(0)) + 0 = S(S(0))
```

The rest of `Nat` left as an exercise...

Right identity - the pattern

- Proof for `S(O)` used proof for `O`
- Proof for `S(S(O))` used proof for `S(O)`
- At each step we look at the last one and think "just one more"
- Exploit the pattern, use a little abstraction.

```
  P(O)    ->  P(S(O))    ->     P(S(S(O))) ...
 /      \    /          \         /
O        S(O)            S(S(O))               ...

   ... P(x)   ->  P(S(x))    ->    P(S(S(x)))
          \     /          \          /
   ...        S(x)           S(S(x))
```

Right identity - the details

- Show that if `x + 0 = x` then `S(x) + 0 = S(x)`
  - Captures the "just one more" part of the proof
- Need to prove the base case as well.
  - In this case the base case is `0 + 0 = 0`
  - So the "just one more" part has somewhere sensible to start from.
  - Otherwise we can also prove that `x + S(0) = x` for $x > O$

Induction

To prove `P(n)` for all `n` $\in$ `Nat`

- prove that `P(0)` holds
- prove that `P(n)` $\rightarrow$ `P(S n)` holds

To prove `P(xs)` for all `xs` $\in$ `[a]`

- prove that `P([])` holds
- prove that `P(xs)` $\rightarrow$ `P(x:xs)`

Similar for trees, any other recursive datatypes

Equational Reasoning

Monoid - Left Identity

Goal : ∀ `bs :: [a]`

```
  [] ++ bs = bs
```

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

Case: `[]`

Monoid - Left Identity

Goal : ∀ `bs :: [a]`

```
[] ++ bs = bs
```

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

Case: `[]`

Monoid - Left Identity

Goal : ∀ `bs :: [a]`

```
[] ++ bs = bs
```

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

Case: `[]`

```
[] ++ bs = bs
```

Monoid - Left Identity

Goal : ∀ `bs :: [a]`

```
[] ++ bs = bs
```

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

Case: `[]`

```
[] ++ bs = bs
```

Monoid - Left Identity

Goal : ∀ `bs :: [a]`

```
[] ++ bs = bs
```

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

Case: `[]`

```
[] ++ bs = bs
```

Monoid - Left Identity

Goal : ∀ bs :: [a]

```
  [] ++ bs = bs
```

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

Case: `[]`

```
[] ++ bs = bs
```

Monoid - Left Identity

Goal : ∀ `bs :: [a]`

```
  [] ++ bs = bs
```

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

Case: `[]`

```
bs = bs
```

Monoid - Left Identity

Goal : ∀ `bs :: [a]`

```
  [] ++ bs = bs
```

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

Case: `[]`

```
    bs = bs
```

Monoid - Left Identity

Goal : ∀ `bs :: [a]`

```
  [] ++ bs = bs
```

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

Case: `[]` ✓

```
bs = bs
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`

```
bs ++ [] = bs
```

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

Cases: `bs = []` and `bs = (x:xs)`

Monoid - Right Identity

Goal : ∀ `bs :: [a]`
`bs ++ [] = bs`

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

Case: `bs = []`

Monoid - Right Identity

Goal : ∀ `bs :: [a]`
`bs ++ [] = bs`

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

Case: `bs = []`

```
bs ++ [] = bs
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`

```
  bs ++ [] = bs
```

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

Case: `bs = []`

```
bs ++ [] = bs
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`
`bs ++ [] = bs`

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

Case: `bs = []`

```
[] ++ [] = []
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`
`bs ++ [] = bs`

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

Case: `bs = []`

```
[] ++ [] = []
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`

```
bs ++ [] = bs
```

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

Case: `bs = []`

```
[] ++ [] = []
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`
`bs ++ [] = bs`

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

Case: `bs = []`

```
[] ++ [] = []
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`

```
  bs ++ [] = bs
```

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

Case: `bs = []`

```
[] = []
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`
`bs ++ [] = bs`

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

Case: `bs = []`

`[] = []`

Monoid - Right Identity

Goal : ∀ `bs :: [a]`

```
  bs ++ [] = bs
```

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

Case: `bs = []` ✓

```
[] = []
```

Monoid - Right Identity

Goal : $\forall$ `bs :: [a]`
```
  bs ++ [] = bs
```

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

Monoid - Right Identity

Goal : ∀ bs :: [a]
 bs ++ [] = bs

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

```
bs      ++ [] = bs
```

Monoid - Right Identity

Goal : ∀ bs :: [a]
  bs ++ [] = bs

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

```
bs     ++ [] = bs
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`

```
  bs ++ [] = bs
```

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

```
(x:xs) ++ [] = (x:xs)
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`

```
bs ++ [] = bs
```

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

```
(x:xs) ++ [] = (x:xs)
```

Monoid - Right Identity

Goal : ∀ bs :: [a]

```
  bs ++ [] = bs
```

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

```
(x:xs) ++ [] = (x:xs)
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`

```
bs ++ [] = bs
```

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

```
(x:xs) ++ [] = (x:xs)
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`

```
bs ++ [] = bs
```

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

```
x:(xs ++ []) = (x:xs)
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`
`bs ++ [] = bs`

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

```
x:(xs ++ []) = (x:xs)
```

Monoid - Right Identity

Goal : ∀ bs :: [a]
  bs ++ [] = bs

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

```
x:(xs ++ []) = (x:xs)
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`
`bs ++ [] = bs`

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

`x:(xs ++ []) = (x:xs)`

Monoid - Right Identity

Goal : ∀ `bs :: [a]`
`bs ++ [] = bs`

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

```
x:(xs)        = (x:xs)
```

Monoid - Right Identity

Goal : ∀ bs :: [a]
  bs ++ [] = bs

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

Case: `bs = (x:xs)`

Assume: `xs ++ [] = xs`

```
(x:xs)       = (x:xs)
```

Monoid - Right Identity

Goal : ∀ `bs :: [a]`
`bs ++ [] = bs`

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

Case: `bs = (x:xs)` ✓

Assume: `xs ++ [] = xs`

```
(x:xs)        = (x:xs)
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Cases: `bs = []` and `bs = (x:xs)`

Monoid - Associativity

Goal : $\forall$ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = []`

Monoid - Associativity

Goal : ∀ bs, cs, ds :: [a]
  bs ++ (cs ++ ds) = (bs ++ cs) ++ ds

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

Case: bs = []

```
bs ++ (cs ++ ds) = (bs ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = []`

`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = []`

```
[] ++ (cs ++ ds) = ([] ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = []`

```
[] ++ (cs ++ ds) = ([] ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`

```
  bs ++ (cs ++ ds) = (bs ++ cs) ++ ds
```

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

Case: `bs = []`

```
[] ++ (cs ++ ds) = ([] ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = []`

```
[] ++ (cs ++ ds) = ([] ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = []`

```
cs ++ ds  = ([] ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = []`

```
cs ++ ds  = ([] ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`

```
  bs ++ (cs ++ ds) = (bs ++ cs) ++ ds
```

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

Case: `bs = []`

```
cs ++ ds  = ([] ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ bs, cs, ds :: [a]
  bs ++ (cs ++ ds) = (bs ++ cs) ++ ds

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

Case: `bs = []`

```
cs ++ ds  = ([] ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ bs, cs, ds :: [a]
  bs ++ (cs ++ ds) = (bs ++ cs) ++ ds

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

Case: `bs = []`

```
cs ++ ds  = (cs)       ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = []`

```
cs ++ ds  =  cs          ++ ds
```

Monoid - Associativity

Goal : $\forall$ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = []` ✓

```
cs ++ ds  =  cs          ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

```
bs ++ (cs ++ ds) =      (bs ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

```
bs ++ (cs ++ ds) =     (bs ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

```
(x:xs) ++ (cs ++ ds) = ((x:xs) ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

```
(x:xs) ++ (cs ++ ds) = ((x:xs) ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

`(x:xs) ++ (cs ++ ds) = ((x:xs) ++ cs) ++ ds`

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

`(x:xs) ++ (cs ++ ds) = ((x:xs) ++ cs) ++ ds`

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

`x:(xs ++ (cs ++ ds)) = ((x:xs) ++ cs) ++ ds`

Monoid - Associativity

```
Goal : ∀ bs, cs, ds :: [a]
  bs ++ (cs ++ ds) = (bs ++ cs) ++ ds

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

```
x:(xs ++ (cs ++ ds)) = ((x:xs) ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

`x:(xs ++ (cs ++ ds)) = ((x:xs) ++ cs) ++ ds`

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`

```
  bs ++ (cs ++ ds) = (bs ++ cs) ++ ds
```

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

```
x:(xs ++ (cs ++ ds)) = ((x:xs) ++ cs) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
```
  bs ++ (cs ++ ds) = (bs ++ cs) ++ ds
```

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

```
x:(xs ++ (cs ++ ds)) = (x:(xs ++ cs)) ++ ds
```

Monoid - Associativity

Goal : ∀ bs, cs, ds :: [a]
  bs ++ (cs ++ ds) = (bs ++ cs) ++ ds

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

Case: bs = (x:xs)

Assume: xs ++ (cs ++ ds) = (xs ++ cs) ++ ds

x:(xs ++ (cs ++ ds)) = (x:(xs ++ cs)) ++ ds

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

```
x:(xs ++ (cs ++ ds)) = (x:(xs ++ cs)) ++ ds
```

Monoid - Associativity

```
Goal : ∀ bs, cs, ds :: [a]
  bs ++ (cs ++ ds) = (bs ++ cs) ++ ds

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

```
x:(xs ++ (cs ++ ds)) = (x:(xs ++ cs)) ++ ds
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

`x:(xs ++ (cs ++ ds)) = x:((xs ++ cs) ++ ds)`

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`
Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

```
x:(xs ++ (cs ++ ds)) = x:((xs ++ cs) ++ ds)
```

Monoid - Associativity

Goal : ∀ bs, cs, ds :: [a]
  bs ++ (cs ++ ds) = (bs ++ cs) ++ ds

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

`x:(xs ++ (cs ++ ds)) = x:((xs ++ cs) ++ ds)`

Monoid - Associativity

```
Goal : ∀ bs, cs, ds :: [a]
  bs ++ (cs ++ ds) = (bs ++ cs) ++ ds

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

```
x:(xs ++ (cs ++ ds)) = x:((xs ++ cs) ++ ds)
```

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

`x:((xs ++ cs) ++ ds) = x:((xs ++ cs) ++ ds)`

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)`

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

`x:((xs ++ cs) ++ ds) = x:((xs ++ cs) ++ ds)`

Monoid - Associativity

Goal : ∀ `bs, cs, ds :: [a]`
`bs ++ (cs ++ ds) = (bs ++ cs) ++ ds`

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

Case: `bs = (x:xs)` ✓

Assume: `xs ++ (cs ++ ds) = (xs ++ cs) ++ ds`

`x:((xs ++ cs) ++ ds) = x:((xs ++ cs) ++ ds)`

Monoid - Associativity - Bird style

```
    bs ++ (cs ++ ds) = (bs ++ cs) ++ ds
                  bs = x:xs
(x:xs) ++ (cs ++ ds) = ((x:xs) ++ cs) ++ ds
              second rule for ++
x:(xs ++ (cs ++ ds)) = ((x:xs) ++ cs) ++ ds
              second rule for ++
x:(xs ++ (cs ++ ds)) = (x:(xs ++ cs)) ++ ds
              second rule for ++
x:(xs ++ (cs ++ ds)) = x:((xs ++ cs) ++ ds)
             inductive hypothesis
x:((xs ++ cs) ++ ds) = x:((xs ++ cs) ++ ds)
```

Functor - Identity

Goal : ∀ `bs :: [a]`
`map id bs = bs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Cases: `bs = []` and `bs = (x:xs)`

Functor - Identity

Goal : ∀ `bs :: [a]`

```
map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

Functor - Identity

Goal : ∀ `bs :: [a]`

```
map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map id bs = bs
```

Functor - Identity

Goal : $\forall$ `bs :: [a]`
`map id bs = bs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map id bs = bs
```

Functor - Identity

Goal : $\forall$ `bs :: [a]`
`map id bs = bs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map id [] = []
```

Functor - Identity

Goal : ∀ `bs :: [a]`

```
  map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map id [] = []
```

Functor - Identity

Goal : $\forall$ bs :: [a]

```
  map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map id [] = []
```

Functor - Identity

Goal : ∀ `bs :: [a]`

```
map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map id [] = []
```

Functor - Identity

Goal : ∀ `bs :: [a]`
`map id bs = bs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
[] = []
```

Functor - Identity

Goal : $\forall$ `bs :: [a]`

```
map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

`[] = []`

Functor - Identity

Goal : ∀ `bs :: [a]`

```
  map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []` ✓

```
[] = []
```

Functor - Identity

Goal : ∀ `bs :: [a]`

```
map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

Functor - Identity

Goal : ∀ `bs :: [a]`

```
map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
map id      bs        = bs
```

Functor - Identity

Goal : ∀ `bs :: [a]`
```
  map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
map id      bs       = bs
```

Functor - Identity

Goal : ∀ `bs :: [a]`
`map id bs = bs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
map id (x : xs)     = (x : xs)
```

Functor - Identity

Goal : ∀ `bs :: [a]`

```
  map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
map id (x : xs)     = (x : xs)
```

Functor - Identity

Goal : ∀ `bs :: [a]`

```
  map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
map id (x : xs)     = (x : xs)
```

Functor - Identity

Goal : ∀ `bs :: [a]`
`map id bs = bs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
map id (x : xs)     = (x : xs)
```

Functor - Identity

Goal : ∀ `bs :: [a]`
`map id bs = bs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
id x : map id xs    = (x : xs)
```

Functor - Identity

Goal : ∀ `bs :: [a]`

```
map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
id x : map id xs   = (x : xs)
```

Functor - Identity

Goal : ∀ `bs :: [a]`

```
map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
x      : map id xs    = (x : xs)
```

Functor - Identity

Goal : ∀ `bs :: [a]`
`map id bs = bs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
x     : map id xs    = (x : xs)
```

Functor - Identity

Goal : ∀ `bs :: [a]`

```
map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
x     : map id xs    = (x : xs)
```

Functor - Identity

Goal : $\forall$ `bs :: [a]`

`map id bs = bs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
x     : map id xs    = (x : xs)
```

Functor - Identity

Goal : ∀ `bs :: [a]`

```
  map id bs = bs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
x     : xs          = (x : xs)
```

Functor - Identity

Goal : ∀ `bs :: [a]`
`map id bs = bs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map id xs = xs`

```
(x     : xs)         = (x : xs)
```

Functor - Identity

Goal : ∀ `bs :: [a]`
`map id bs = bs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)` ✓
Assume: `map id xs = xs`

```
(x     : xs)          = (x : xs)
```

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Cases: `bs = []` and `bs = (x:xs)`

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

Functor - Composition

Goal : ∀ bs :: [a], f :: a → b, g :: b → c
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g (map f bs) = map (g . f) bs
```

Functor - Composition

Goal : $\forall$ `bs :: [a], f :: a` $\rightarrow$ `b, g :: b` $\rightarrow$ `c`

```
map g (map f bs) = map (g . f) xs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g (map f bs) = map (g . f) bs
```

Functor - Composition

Goal : $\forall$ `bs :: [a], f :: a` $\rightarrow$ `b, g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g (map f []) = map (g . f) []
```

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g (map f []) = map (g . f) []
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\rightarrow$ b, g :: b $\rightarrow$ c
map g (map f bs) = map (g . f) xs

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g (map f []) = map (g . f) []
```

Functor - Composition

Goal : $\forall$ `bs :: [a], f :: a` $\rightarrow$ `b, g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g (map f []) = map (g . f) []
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\rightarrow$ b, g :: b $\rightarrow$ c
  map g (map f bs) = map (g . f) xs

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g (map f []) =              []
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\rightarrow$ b, g :: b $\rightarrow$ c
map g (map f bs) = map (g . f) xs

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g (map f []) =              []
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\rightarrow$ b, g :: b $\rightarrow$ c
  map g (map f bs) = map (g . f) xs

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g (map f [])  =            []
```

Functor - Composition

Goal : $\forall$ `bs :: [a], f :: a` $\rightarrow$ `b, g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g (map f [])  =            []
```

Functor - Composition

Goal : ∀ bs :: [a], f :: a → b, g :: b → c
map g (map f bs) = map (g . f) xs

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g          [] =            []
```

Functor - Composition

Goal : $\forall$ `bs :: [a], f :: a` $\rightarrow$ `b, g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g          [] =          []
```

Functor - Composition

Goal : $\forall$ `bs :: [a], f :: a` $\rightarrow$ `b, g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g [] = []
```

Functor - Composition

Goal : $\forall$ `bs :: [a], f :: a` $\rightarrow$ `b, g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
map g            [] =             []
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\rightarrow$ b, g :: b $\rightarrow$ c
map g (map f bs) = map (g . f) xs

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

`[] = []`

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`

```
map g (map f bs) = map (g . f) xs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []`

```
[] = []
```

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`

```
map g (map f bs) = map (g . f) xs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = []` ✓

```
[] =                    []
```

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`

```
map g (map f bs) = map (g . f) xs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

Functor - Composition

Goal : ∀ bs :: [a], f :: a → b, g :: b → c

```
map g (map f bs) = map (g . f) xs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
map g (map f bs)        = map (g . f) bs
```

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`

```
map g (map f bs) = map (g . f) xs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
map g (map f bs)        = map (g . f) bs
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\rightarrow$ b, g :: b $\rightarrow$ c
  map g (map f bs) = map (g . f) xs

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
map g (map f (x:xs))    = map (g . f) (x:xs)
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\rightarrow$ b, g :: b $\rightarrow$ c
  map g (map f bs) = map (g . f) xs

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
map g (map f (x:xs))   = map (g . f) (x:xs)
```

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\to$ `b`, `g :: b` $\to$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`
Assume: `map g (map f xs) = map (g . f) xs`

```
map g (map f (x:xs))    = map (g . f) (x:xs)
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\to$ b, g :: b $\to$ c
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
map g (map f (x:xs))    = map (g . f) (x:xs)
```

Functor - Composition

Goal : $\forall$ `bs :: [a], f :: a` $\to$ `b, g :: b` $\to$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
map g (map f (x:xs))   = (g . f) x : map (g . f) xs
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\rightarrow$ b, g :: b $\rightarrow$ c
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
map g (map f (x:xs))    = (g . f) x : map (g . f) xs
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\to$ b, g :: b $\to$ c
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
map g (map f (x:xs))    = (g . f) x : map (g . f) xs
```

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
map g (map f (x:xs))    = (g . f) x : map (g . f) xs
```

Functor - Composition

Goal : $\forall$ `bs :: [a], f :: a` $\rightarrow$ `b, g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

`map g (f x : map f xs) = (g . f) x : map (g . f) xs`

Functor - Composition

Goal : ∀ bs :: [a], f :: a → b, g :: b → c
  map g (map f bs) = map (g . f) xs

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
map g (f x : map f xs) = (g . f) x : map (g . f) xs
```

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

`map g (f x : map f xs) = (g . f) x : map (g . f) xs`

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`

```
  map g (map f bs) = map (g . f) xs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
map g (f x : map f xs) = (g . f) x : map (g . f) xs
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\to$ b, g :: b $\to$ c
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
g (f x) : map g (map f xs) = (g . f) x : map (g . f) xs
```

Functor - Composition

Goal : ∀ `bs :: [a]`, `f :: a` → `b`, `g :: b` → `c`
```
map g (map f bs) = map (g . f) xs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
g (f x) : map g (map f xs) = (g . f) x : map (g . f) xs
```

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`

```
map g (map f bs) = map (g . f) xs
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
(g . f) x : map g (map f xs) = (g . f) x : map (g . f) xs
```

Functor - Composition

Goal : $\forall$ `bs :: [a], f :: a` $\rightarrow$ `b, g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

`(g . f) x : map g (map f xs) = (g . f) x : map (g . f) xs`

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

`(g . f) x : map g (map f xs) = (g . f) x : map (g . f) xs`

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\rightarrow$ b, g :: b $\rightarrow$ c
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`
Assume: `map g (map f xs) = map (g . f) xs`

```
(g . f) x : map g (map f xs) = (g . f) x : map (g . f) xs
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\rightarrow$ b, g :: b $\rightarrow$ c
  map g (map f bs) = map (g . f) xs

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

```
(g . f) x : map (g . f) xs    = (g . f) x : map (g . f) xs
```

Functor - Composition

Goal : $\forall$ bs :: [a], f :: a $\rightarrow$ b, g :: b $\rightarrow$ c
map g (map f bs) = map (g . f) xs

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)`

Assume: `map g (map f xs) = map (g . f) xs`

`(g . f) x : map (g . f) xs   = (g . f) x : map (g . f) xs`

Functor - Composition

Goal : $\forall$ `bs :: [a]`, `f :: a` $\rightarrow$ `b`, `g :: b` $\rightarrow$ `c`
`map g (map f bs) = map (g . f) xs`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Case: `bs = (x:xs)` ✓
Assume: `map g (map f xs) = map (g . f) xs`

```
(g . f) x : map (g . f) xs    = (g . f) x : map (g . f) xs
```

Coq

To the Coqmobile!

Coq extraction - append

```
module Append where

import qualified Prelude

data List a =
   Nil
 | Cons a (List a)

append :: (List a1) -> (List a1) -> List a1
append xs ys =
  case xs of {
   Nil -> ys;
   Cons h t -> Cons h (append t ys)}
```

Coq extraction - map

```
module Map where

import qualified Prelude

data List a =
   Nil
 | Cons a (List a)

map :: (a1 -> a2) -> (List a1) -> List a2
map f l =
  case l of {
   Nil -> Nil;
   Cons a t -> Cons (f a) (map f t)}
```

Conclusion

- Equational reasoning is easy and useful
- Theorem provers can help if you want to be really sure
- Very few excuses for law-breaking typeclass instances

Other cool stuff

- Bottom, partial data and strictness
- Monad laws
  - Similar to the monoid laws
  - Just a monoid in the category of endofunctors after all
- Fold fusion
  - when does `f . fold g a = fold h b`?
- Program synthesis
- Theorems for free

Books

- How To Prove It - Velleman
- Introduction to Functional Programming with Haskell - Bird
- Interactive Theorem Proving and Program Development - Bertot
- Software Foundations - Pierce

Slides and code

https://github.com/dalaing/bfpg-2013-03