# A Larger Equational Proof on Lists

Principles of Functional Programming

# A Law of Reverse

For a more difficult example, let's consider the `reverse` function.

We pick its inefficient definition, because its more amenable to equational proofs:

```
      Nil.reverse  =  Nil                          // 1st clause
(x :: xs).reverse  =  xs.reverse ++ List(x)        // 2nd clause
```

We'd like to prove the following proposition

```
xs.reverse.reverse  =  xs
```

# Proof

By induction on `xs`. The base case is easy:

```
Nil.reverse.reverse
=   Nil.reverse             // by 1st clause of reverse
=   Nil                     // by 1st clause of reverse
```

# Proof

By induction on `xs`. The base case is easy:

```
Nil.reverse.reverse
=   Nil.reverse            // by 1st clause of reverse
=   Nil                    // by 1st clause of reverse
```

For the induction step, let's try:

```
(x :: xs).reverse.reverse
=   (xs.reverse ++ List(x)).reverse // by 2nd clause of reverse
```

# Proof

By induction on `xs`. The base case is easy:

```
Nil.reverse.reverse
=   Nil.reverse                 // by 1st clause of reverse
=   Nil                         // by 1st clause of reverse
```

For the induction step, let's try:

```
(x :: xs).reverse.reverse
=   (xs.reverse ++ List(x)).reverse // by 2nd clause of reverse
```

We can't do anything more with this expression, therefore we turn to the right-hand side:

```
x :: xs
=   x :: xs.reverse.reverse          // by induction hypothesis
```

Both sides are simplified in different expressions.

# To Do

We still need to show:

```
(xs.reverse ++ List(x)).reverse  =  x :: xs.reverse.reverse
```

Trying to prove it directly by induction doesn't work.

We must instead try to *generalize* the equation. For *any* list `ys`,

```
(ys ++ List(x)).reverse  =  x :: ys.reverse
```

This equation can be proved by a second induction argument on `ys`.

# Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse        // to show: =  x :: Nil.reverse
```

# Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse        // to show: =  x :: Nil.reverse

=   List(x).reverse             // by 1st clause of ++
```

# Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse      // to show: =  x :: Nil.reverse

=   List(x).reverse           // by 1st clause of ++

=   (x :: Nil).reverse        // by definition of List
```

# Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse       // to show: =  x :: Nil.reverse

=   List(x).reverse            // by 1st clause of ++

=   (x :: Nil).reverse         // by definition of List

=   Nil ++ (x :: Nil)          // by 2nd clause of reverse
```

# Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse        // to show: =  x :: Nil.reverse

=   List(x).reverse             // by 1st clause of ++

=   (x :: Nil).reverse          // by definition of List

=   Nil ++ (x :: Nil)           // by 2nd clause of reverse

=   x :: Nil                    // by 1st clause of ++
```

# Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse        // to show: =  x :: Nil.reverse

=   List(x).reverse             // by 1st clause of ++

=   (x :: Nil).reverse          // by definition of List

=   Nil ++ (x :: Nil)           // by 2nd clause of reverse

=   x :: Nil                    // by 1st clause of ++

=   x :: Nil.reverse            // by 1st clause of reverse
```

# Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse          // to show: =  x :: (y :: ys).reverse
```

# Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse            // to show: =  x :: (y :: ys).reverse

=   (y :: (ys ++ List(x))).reverse        // by 2nd clause of ++
```

# Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse          // to show: =  x :: (y :: ys).reverse

=    (y :: (ys ++ List(x))).reverse     // by 2nd clause of ++

=    (ys ++ List(x)).reverse ++ List(y) // by 2nd clause of reverse
```

# Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse          // to show: =  x :: (y :: ys).reverse

=   (y :: (ys ++ List(x))).reverse      // by 2nd clause of ++

=   (ys ++ List(x)).reverse ++ List(y)  // by 2nd clause of reverse

=   (x :: ys.reverse) ++ List(y)        // by the induction hypothesis
```

# Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse          // to show: =  x :: (y :: ys).reverse

=   (y :: (ys ++ List(x))).reverse      // by 2nd clause of ++

=   (ys ++ List(x)).reverse ++ List(y)  // by 2nd clause of reverse

=   (x :: ys.reverse) ++ List(y)        // by the induction hypothesis

=   x :: (ys.reverse ++ List(y))        // by 1st clause of ++
```

# Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse          // to show: =  x :: (y :: ys).reverse
=   (y :: (ys ++ List(x))).reverse      // by 2nd clause of ++
=   (ys ++ List(x)).reverse ++ List(y)  // by 2nd clause of reverse
=   (x :: ys.reverse) ++ List(y)        // by the induction hypothesis
=   x :: (ys.reverse ++ List(y))        // by 1st clause of ++
=   x :: (y :: ys).reverse              // by 2nd clause of reverse
```

This establishes the auxiliary equation, and with it the main proposition.

# Exercise

Prove the following distribution law for `map` over concatenation.

For any lists `xs`, `ys`, function `f`:

```
(xs ++ ys).map(f)  =  xs.map(f) ++ ys.map(f)
```

You will need the clauses of $++$ as well as the following clauses for `map`:

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