香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Principles of Programming Languages (Lecture 7)

COMP 3031, Fall 2025

Lionel Parreaux

Reasoning About Lists

Laws of Concat

Recall the concatenation operation `:::` on lists.

We would like to verify that concatenation is associative, and that it admits the empty list `Nil` as neutral element to the left and to the right:

```
(xs ::: ys) ::: zs  =  xs ::: (ys ::: zs)
       xs ::: Nil  =  xs
       Nil ::: xs  =  xs
```

Q: How can we prove properties like these?

Laws of Concat

Recall the concatenation operation `:::` on lists.

We would like to verify that concatenation is associative, and that it admits the empty list `Nil` as neutral element to the left and to the right:

```
(xs ::: ys) ::: zs  =  xs ::: (ys ::: zs)
       xs ::: Nil  =  xs
       Nil ::: xs  =  xs
```

Q: How can we prove properties like these?

A: By *structural induction* on lists.

Reminder: Natural Induction

Recall the principle of proof by *natural induction*:

To show a property $P(n)$ for all the integers $n \geq b$,

- Show that we have $P(b)$ (*base case*),
- for all integers $n \geq b$ show the *induction step*:
  if one has $P(n)$, *then one also has* $P(n+1)$.

Example

Given:

```
def factorial(n: Int): Int =
  if n == 0 then 1         // 1st clause
  else n * factorial(n-1)  // 2nd clause
```

Show that, for all `n >= 4`

```
factorial(n) >= power(2, n)
```

Base Case

Base case: 4

This case is established by simple calculations:

```
factorial(4) = 24 >= 16 = power(2, 4)
```

Induction Step

Induction step: `n+1`

We have for `n >= 4`:

```
factorial(n + 1)
```

Induction Step

Induction step: `n+1`

We have for `n >= 4`:

```
factorial(n + 1)

=  (n + 1) * factorial(n)    // by 2nd clause in factorial
```

Induction Step

Induction step: `n+1`

We have for `n >= 4`:

```
factorial(n + 1)

=  (n + 1) * factorial(n)    // by 2nd clause in factorial

>   2 * factorial(n)         // by calculating
```

Induction Step

Induction step: n+1

We have for `n >= 4`:

```
factorial(n + 1)

=  (n + 1) * factorial(n)    // by 2nd clause in factorial

>   2 * factorial(n)         // by calculating

>=  2 * power(2, n)          // by induction hypothesis
```

Induction Step

Induction step: `n+1`

We have for `n >= 4`:

```
factorial(n + 1)

=  (n + 1) * factorial(n)    // by 2nd clause in factorial

>   2 * factorial(n)         // by calculating

>=  2 * power(2, n)          // by induction hypothesis

=   power(2, n + 1)          // by definition of power
```

Referential Transparency

Note that a proof can freely apply reduction steps as equalities to some part of a term.

That works because pure functional programs don't have side effects; so that a term is equivalent to the term to which it reduces.

This principle is called *referential transparency*.

Structural Induction

The principle of structural induction is analogous to natural induction:

To prove a property `P(xs)` for all lists `xs`,

- show that `P(Nil)` holds (*base case*),
- for a list `xs` and some element `x`, show the *induction step*:
  if `P(xs)` *holds, then* `P(x :: xs)` *also holds.*

Example

Let's show that, for lists `xs`, `ys`, `zs`:

```
(xs ::: ys) ::: zs  =  xs ::: (ys ::: zs)
```

To do this, use structural induction on `xs`. From the previous implementation of `:::`,

```
extension [T](xs: List[T])
  def ::: (ys: List[T]) = xs match
    case Nil => ys
    case x :: xs1 => x :: (xs1 ::: ys)
```

distill two *defining clauses* of `:::`:

```
      Nil ::: ys  =  ys                      // 1st clause
(x :: xs1) ::: ys  =  x :: (xs1 ::: ys)     // 2nd clause
```

Base Case

$\boxed{\textbf{Base case: } \texttt{Nil}}$

For the left-hand side we have:

```
(Nil ::: ys) ::: zs
```

Base Case

Base case: `Nil`

For the left-hand side we have:

```
(Nil ::: ys) ::: zs

=   ys ::: zs          // by 1st clause of :::
```

Base Case

Base case: `Nil`

For the left-hand side we have:

```
(Nil ::: ys) ::: zs

=   ys ::: zs          // by 1st clause of :::
```

For the right-hand side, we have:

```
Nil ::: (ys ::: zs)
```

Base Case

Base case: `Nil`

For the left-hand side we have:

```
(Nil ::: ys) ::: zs

=   ys ::: zs        // by 1st clause of :::
```

For the right-hand side, we have:

```
Nil ::: (ys ::: zs)

=   ys ::: zs        // by 1st clause of :::
```

This case is therefore established.

Induction Step: LHS

Induction step: `x :: xs`

For the left-hand side, we have:

```
((x :: xs) ::: ys) ::: zs
```

Induction Step: LHS

Induction step: `x :: xs`

For the left-hand side, we have:

```
((x :: xs) ::: ys) ::: zs

=   (x :: (xs ::: ys)) ::: zs    // by 2nd clause of :::
```

Induction Step: LHS

Induction step: `x :: xs`

For the left-hand side, we have:

```
((x :: xs) ::: ys) ::: zs

=   (x :: (xs ::: ys)) ::: zs    // by 2nd clause of :::

=   x :: ((xs ::: ys) ::: zs)    // by 2nd clause of :::
```

Induction Step: LHS

Induction step: `x :: xs`

For the left-hand side, we have:

```
((x :: xs) ::: ys) ::: zs

=   (x :: (xs ::: ys)) ::: zs    // by 2nd clause of :::

=   x :: ((xs ::: ys) ::: zs)    // by 2nd clause of :::

=   x :: (xs ::: (ys ::: zs))    // by induction hypothesis
```

Induction Step: RHS

To make progress on the right-hand side, consider:

```
(x :: xs) ::: (ys ::: zs)
```

Induction Step: RHS

To make progress on the right-hand side, consider:

```
(x :: xs) ::: (ys ::: zs)

=   x :: (xs ::: (ys ::: zs))    // by 2nd clause of :::
```

So this case (the inductive case) is established.

Induction Step: RHS

To make progress on the right-hand side, consider:

```
(x :: xs) ::: (ys ::: zs)

=   x :: (xs ::: (ys ::: zs))    // by 2nd clause of :::
```

So this case (the inductive case) is established.

And with it, so is the property, i.e.:

```
(xs ::: ys) ::: zs  =  xs ::: (ys ::: zs)
```

Exercise

Show by induction on `xs` that `xs ::: Nil  =  xs`.

How many equations do you need for the inductive step?

0 2

0 3

0 4

Exercise

Show by induction on `xs` that `xs ::: Nil  =  xs`.

How many equations do you need for the inductive step?

X	2
0	3
0	4

A Larger Equational Proof on Lists

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 ::: x :: Nil      // 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 ::: x :: Nil).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 ::: x :: Nil).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 into *different* expressions.

To Do

We still need to show:

```
(xs.reverse ::: x :: Nil).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 ::: x :: Nil).reverse  =  x :: ys.reverse
```

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

Auxiliary Equation, Base Case

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

Auxiliary Equation, Base Case

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

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

Auxiliary Equation, Base Case

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

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

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

Auxiliary Equation, Base Case

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

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

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

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

Auxiliary Equation, Base Case

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

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

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

Auxiliary Equation, Inductive Step

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

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

Auxiliary Equation, Inductive Step

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

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

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

Auxiliary Equation, Inductive Step

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

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

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

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

Auxiliary Equation, Inductive Step

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

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

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

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

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

Auxiliary Equation, Inductive Step

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

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

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

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

=   x :: (ys.reverse ::: y :: Nil)          // by 2nd 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)
```

Other Collections

Other Sequences

We have seen that lists are *linear*: Access to the first element is much faster than access to the middle or end of a list.

The Scala library also defines an alternative sequence implementation, `Vector`.

This one has more evenly balanced access patterns than `List`.

Operations on Vectors

Vectors are created analogously to lists:

```
val nums = Vector(1, 2, 3, -88)
val people = Vector("Bob", "James", "Peter")
```

They support the same operations as lists, with the exception of `::`

Instead of `x :: xs`, there is

`x +: xs`	Create a new vector with leading element `x`, followed by all elements of `xs`.
`xs :+ x`	Create a new vector with trailing element `x`, preceded by all elements of `xs`.

(Note that the ‘`:`’ always points to the sequence.)

Collection Hierarchy

A common base class of `List` and `Vector` is `Seq`, the class of all *sequences*.

`Seq` itself is a subclass of `Iterable`.

```
                 Iterable
               /    |     \
              /     |      \
  .............Seq     Set      Map
  .          /   \
  .         /     \
Array    List     Vector
```

Arrays and Strings

Arrays and Strings support the same operations as `Seq` and can implicitly be converted to sequences where needed.

(They cannot be subclasses of `Seq` because they come from Java)

```
val xs: Array[Int] = Array(1, 2, 3)
xs.map(x => 2 * x)

val ys: String = "Hello world!"
ys.filter(_.isUpper)
```

Ranges

Another simple kind of sequence is the *range*.

It represents a sequence of evenly spaced integers.

Three operators:

`to` (inclusive), `until` (exclusive), `by` (to determine step value):

```
val r: Range = 1 until 5
val s: Range = 1 to 5
1 to 10 by 3
6 to 1 by -2
```

A `Range` is represented as a single object with three fields: lower bound, upper bound, step value.

Some more Sequence Operations:

`xs.exists(p)`	`true` if there is an element `x` of `xs` such that `p(x)` holds, `false` otherwise.
`xs.forall(p)`	`true` if `p(x)` holds for all elements `x` of `xs`, `false` otherwise.
`xs.zip(ys)`	A sequence of pairs drawn from corresponding elements of sequences `xs` and `ys`.
`xs.unzip`	Splits a sequence of pairs `xs` into two sequences consisting of the first, respectively second halves of all pairs.
`xs.flatMap(f)`	Applies collection-valued function `f` to all elements of `xs` and concatenates the results
`xs.sum`	The sum of all elements of this numeric collection.
`xs.product`	The product of all elements of this numeric collection
`xs.max`	The maximum of all elements of this collection (an `Ordering` must exist)
`xs.min`	The minimum of all elements of this collection

Example: Combinations

To list all combinations of numbers `x` and `y` where `x` is drawn from `1..M` and `y` is drawn from `1..N`:

```
(1 to M).flatMap(x =>
```

Example: Combinations

To list all combinations of numbers `x` and `y` where `x` is drawn from `1..M` and `y` is drawn from `1..N`:

```
(1 to M).flatMap(x => (1 to N).map(y => (x, y)))
```

Example: Scalar Product

To compute the scalar product of two vectors:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map((x, y) => x * y).sum
```

Example: Scalar Product

To compute the scalar product of two vectors:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map((x, y) => x * y).sum
```

Note that there is some automatic decomposition going on here.

Each pair of elements from `xs` and `ys` is split into its halves which are then passed as the `x` and `y` parameters to the lambda.

Example: Scalar Product

If we wanted to be more explicit, we could also write scalar product like this:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map(xy => xy._1 * xy._2).sum
```

Example: Scalar Product

On the other hand, if we wanted to be more even more concise, we could also write it like this:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map(_ * _).sum
```

Exercise:

A number `n` is *prime* if the only divisors of `n` are `1` and `n` itself.

What is a high-level way to write a test for primality of numbers? For once, value conciseness over efficiency.

```
def isPrime(n: Int): Boolean = ???
```

Exercise:

A number `n` is *prime* if the only divisors of `n` are `1` and `n` itself.

What is a high-level way to write a test for primality of numbers? For once, value conciseness over efficiency.

```
def isPrime(n: Int): Boolean =
  (2 to n - 1).forall(d => n % d != 0)
```

Maps

Map

Another fundamental collection type is the *map*.

A map of type `Map[Key, Value]` is a data structure that associates keys of type `Key` with values of type `Value`.

Examples:

```
val romanNumerals = Map("I" -> 1, "V" -> 5, "X" -> 10)
val capitalOfCountry = Map("US" -> "Washington", "Switzerland" -> "Bern")
```

Maps are Iterables

Class `Map[Key, Value]` extends the collection type `Iterable[(Key, Value)]`.

Therefore, maps support the same collection operations as other iterables do. Example:

```
val countryOfCapital = capitalOfCountry.map((x, y) => (y, x))
                       // Map(”Washington” -> ”US”, ”Bern” -> ”Switzerland”)
```

Note that maps extend iterables of key/value *pairs*.

In fact, the syntax `key -> value` is just an alternative way to write the pair `(key, value)`. (it is implemented as an extension method in `Predef`).

Maps are Functions

Class `Map[Key, Value]` also extends the function type `Key => Value`, so maps can be used everywhere functions can.

In particular, maps can be applied to key arguments:

```
capitalOfCountry("US")          // "Washington"
```

Querying Map

Caution: Applying a map to a non-existing key raises an exception:

```
capitalOfCountry("Andorra")
  // java.util.NoSuchElementException: key not found: Andorra
```

To query a map without knowing beforehand whether it contains a given key, you can use the `get` operation:

```
capitalOfCountry.get("US")      // Some("Washington")
capitalOfCountry.get("Andorra") // None
```

The result of a `get` operation is an `Option` value.

The `Option` Type

The `Option` type is defined as:

```
sealed abstract class Option[+A]

case class Some[+A](value: A) extends Option[A]
object None extends Option[Nothing]
```

The expression `map.get(key)` returns

- `None` if `map` does not contain the given `key`,
- `Some(x)` if `map` associates the given `key` with the value `x`.

Decomposing Option

Since options are defined as case classes, they can be decomposed using pattern matching:

```
def showCapital(country: String) = capitalOfCountry.get(country) match
  case Some(capital) => capital
  case None => "missing data"

showCapital("US")      // "Washington"
showCapital("Andorra") // "missing data"
```

Options also support quite a few operations of the other collections.

We invite you to try them out!

Updating Maps

Functional updates of a map are done with the + and ++ operations:

`m + (k -> v)`	The map that takes key ‘k‘ to value ‘v‘ and is otherwise equal to ‘m‘
`m ++ kvs`	The map ‘m‘ updated via ‘+‘ with all key/value pairs in ‘kvs‘

These operations are purely functional. For instance,

val m1 = Map("red" -> 1, "blue" -> 2)	> *m1 = Map(red -> 1, blue -> 2)*
val m2 = m1 + ("blue" -> 3)	> *m2 = Map(red -> 1, blue -> 3)*
m1	> *Map(red -> 1, blue -> 2)*

Sorted and GroupBy

Two useful operations known from SQL queries are `group by` and `order by`.

`order by` on a Scala collection can be expressed using `sortWith` and `sorted`:

```
val fruit = List("apple", "pear", "orange", "pineapple")
fruit.sortWith(_.length < _.length)  // List("pear", "apple", "orange", "pineappl
fruit.sorted                         // List("apple", "orange", "pear", "pineappl
```

`groupBy` is available on Scala collections. It partitions a collection into a map of collections according to a *discriminator function* `f`.

Example:

```
fruit.groupBy(_.head)   //> Map(p -> List(pear, pineapple),
                        //|     a -> List(apple),
                        //|     o -> List(orange))
```

Map Example

A polynomial can be seen as a map from exponents to coefficients.

For instance, $x^3 - 2x + 5$ can be represented with the map.

```
Map(0 -> 5, 1 -> -2, 3 -> 1)
```

Based on this observation, let's design a class `Polynom` that represents polynomials as maps.

Default Values

So far, maps were *partial functions*: Applying a map to a key value in `map(key)` could lead to an exception, if the key was not stored in the map.

There is an operation `withDefaultValue` that turns a map into a total function:

```
val cap1 = capitalOfCountry.withDefaultValue("<unknown>")
cap1("Andorra")                // "<unknown>"
```

Variable Length Argument Lists

It's quite inconvenient to have to write

```
Polynom(Map(1 -> 2.0, 3 -> 4.0, 5 -> 6.2))
```

Can one do without the `Map(...)`?

Problem: The number of `key -> value` pairs passed to `Map` can vary.

Variable Length Argument Lists

It's quite inconvenient to have to write

```
Polynom(Map(1 -> 2.0, 3 -> 4.0, 5 -> 6.2))
```

Can one do without the `Map(...)`?

Problem: The number of `key -> value` pairs passed to `Map` can vary.

We can accommodate this pattern using a *repeated parameter*:

```
def Polynom(bindings: (Int, Double)*) =
  Polynom(bindings.toMap.withDefaultValue(0))

Polynom(1 -> 2.0, 3 -> 4.0, 5 -> 6.2)
```

Inside the `Polynom` function, `bindings` is seen as a `Seq[(Int, Double)]`.

Final Implementation of Polynom

```
class Polynom(nonZeroTerms: Map[Int, Double]):
  def this(bindings: (Int, Double)*) = this(bindings.toMap)

  val terms = nonZeroTerms.withDefaultValue(0.0)
  def + (other: Polynom) =
    Polynom(terms ++ other.terms.map((exp, coeff) => (exp, terms(exp) + coeff)))

  override def toString = if terms.isEmpty then "0" else
    val termStrings =
      terms.toList.sorted.reverse.map: (exp, coeff) =>
        val exponent = if exp == 0 then "" else s"x^$exp"
        s"$coeff$exponent"
    termStrings.mkString(" + ")
```

Exercise

The + operation on `Polynom` used map concatenation with `++`. Design another version of + in terms of `foldLeft`:

```
def + (other: Polynom) =
  Polynom(other.terms.foldLeft(???)(addTerm))

def addTerm(terms: Map[Int, Double], term: (Int, Double)) =
  ???
```

Exercise

The + operation on `Polynom` used map concatenation with `++`. Design another version of + in terms of `foldLeft`:

```
def + (other: Polynom) =
  Polynom(other.terms.foldLeft(terms)(addTerm))

def addTerm(terms: Map[Int, Double], term: (Int, Double)) =
  val (exp, coeff) = term
  terms + (exp, coeff + terms(exp))
```

Which of the two versions do you believe is more efficient?

```
  O         The version using ++
  O         The version using foldLeft
```

Exercise

The + operation on `Polynom` used map concatenation with `++`. Design another version of + in terms of `foldLeft`:

```
def + (other: Polynom) =
  Polynom(other.terms.foldLeft(terms)(addTerm))

def addTerm(terms: Map[Int, Double], term: (Int, Double)) =
  val (exp, coeff) = term
  terms + (exp, coeff + terms(exp))
```

Which of the two versions do you believe is more efficient?

```
O         The version using ++
X         The version using foldLeft
```