A Closer Look At Lists

Principles of Functional Programming

Lists Recap

Lists are the core data structure we will work with over the next weeks.

Type: `List[Fruit]`

Construction:

```
val fruits = List("Apple", "Orange", "Banana")
val nums = 1 :: 2 :: Nil
```

Decomposition:

```
fruits.head      // "Apple"
nums.tail        // 2 :: Nil
nums.isEmpty     // false

nums match
  case x :: y :: _ => x + y    // 3
```

List Methods (1)

Sublists and element access:

`xs.length`	The number of elements of `xs`.
`xs.last`	The list's last element, exception if `xs` is empty.
`xs.init`	A list consisting of all elements of `xs` except the last one, exception if `xs` is empty.
`xs.take(n)`	A list consisting of the first `n` elements of `xs`, or `xs` itself if it is shorter than `n`.
`xs.drop(n)`	The rest of the collection after taking `n` elements.
`xs(n)`	(or, written out, `xs.apply(n)`). The element of `xs` at index `n`.

List Methods (2)

Creating new lists:

`xs ++ ys`	The list consisting of all elements of `xs` followed by all elements of `ys`.
`xs.reverse`	The list containing the elements of `xs` in reversed order.
`xs.updated(n, x)`	The list containing the same elements as `xs`, except at index `n` where it contains `x`.

Finding elements:

`xs.indexOf(x)`	The index of the first element in `xs` equal to `x`, or `-1` if `x` does not appear in `xs`.
`xs.contains(x)`	same as `xs.indexOf(x) >= 0`

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case List() => throw Error("last of empty list")
  case List(x) =>
  case y :: ys =>
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case List() => throw Error("last of empty list")
  case List(x) => x
  case y :: ys =>
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case List() => throw Error("last of empty list")
  case List(x) => x
  case y :: ys => last(ys)
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case List() => throw Error("last of empty list")
  case List(x) => x
  case y :: ys => last(ys)
```

So, `last` takes steps proportional to the length of the list `xs`.

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error("init of empty list")
  case List(x) => ???
  case y :: ys => ???
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error(”init of empty list”)
  case List(x) =>
  case y :: ys =>
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error(”init of empty list”)
  case List(x) => List()
  case y :: ys =>
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case List() => throw Error(”init of empty list”)
  case List(x) => List()
  case y :: ys => y :: init(ys)
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

```
extension [T](xs: List[T])
  def ++ (ys: List[T]): List[T] =
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

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

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

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

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

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

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for ++:

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

What is the complexity of concat?

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for `++`:

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

What is the complexity of `concat`?

Answer: `O(xs.length)`

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

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

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

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

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil => Nil
    case y :: ys => ys.reverse ++ List(y)
```

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil => Nil
    case y :: ys => ys.reverse ++ List(y)
```

What is the complexity of `reverse`?

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil => Nil
    case y :: ys => ys.reverse ++ List(y)
```

What is the complexity of `reverse`?

Answer: `O(xs.length * xs.length)`

Can we do better? (to be solved later).

Exercise

Remove the `n`'th element of a list `xs`. If `n` is out of bounds, return `xs` itself.

```
def removeAt[T](n: Int, xs: List[T]) = ???
```

Usage example:

```
removeAt(1, List('a', 'b', 'c', 'd'))
```

> *List(a, c, d)*

Exercise (Harder, Optional)

Flatten a list structure:

```scala
def flatten(xs: List[Any]): List[Any] = ???

flatten(List(List(1, 1), 2, List(3, List(5, 8))))
              >   res0: List[Any] = List(1, 1, 2, 3, 5, 8)
```

Tuples and Generic Methods

Principles of Functional Programming

Sorting Lists Faster

As a non-trivial example, let's design a function to sort lists that is more efficient than insertion sort.

A good algorithm for this is *merge sort*. The idea is as follows:

If the list consists of zero or one elements, it is already sorted.

Otherwise,

- Separate the list into two sub-lists, each containing around half of the elements of the original list.
- Sort the two sub-lists.
- Merge the two sorted sub-lists into a single sorted list.

First MergeSort Implementation

Here is the implementation of that algorithm in Scala:

```scala
def msort(xs: List[Int]): List[Int] =
  val n = xs.length / 2
  if n == 0 then xs
  else
    def merge(xs: List[Int], ys: List[Int]) = ???
    val (fst, snd) = xs.splitAt(n)
    merge(msort(fst), msort(snd))
```

The SplitAt Function

The `splitAt` function on lists returns two sublists

- the elements up the the given index
- the elements from that index

The lists are returned in a *pair*.

Detour: Pair and Tuples

The pair consisting of `x` and `y` is written `(x, y)` in Scala.

Example

```
val pair = ("answer", 42)
```
> *pair : (String, Int) = (answer,42)*

The type of `pair` above is `(String, Int)`.

Pairs can also be used as patterns:

```
val (label, value) = pair
```
> *label: String = answer, value: Int = 42*

This works analogously for tuples with more than two elements.

Translation of Tuples

For small `(*)` n, the tuple type $(\mathsf{T}_1, \ldots, \mathsf{T}_n)$ is an abbreviation of the parameterized type

$$\texttt{scala.Tuple}n[\mathsf{T}_1, \ldots, \mathsf{T}_n]$$

A tuple expression $(\mathsf{e}_1, \ldots, \mathsf{e}_n)$ is equivalent to the function application

$$\texttt{scala.Tuple}n(\mathsf{e}_1, \ldots, \mathsf{e}_n)$$

A tuple pattern $(\mathsf{p}_1, \ldots, \mathsf{p}_n)$ is equivalent to the constructor pattern

$$\texttt{scala.Tuple}n(\mathsf{p}_1, \ldots, \mathsf{p}_n)$$

`(*)` Currently, "small" = up to 22. There's also a TupleXXL class that handles Tuples larger than that limit.

The Tuple class

Here, all `Tuple`*n* classes are modeled after the following pattern:

```
case class Tuple2[T1, T2](_1: +T1, _2: +T2):
  override def toString = "(" + _1 + "," + _2 +")"
```

The fields of a tuple can be accessed with names _1, _2, ...

So instead of the pattern binding

```
val (label, value) = pair
```

one could also have written:

```
val label = pair._1
val value = pair._2
```

But the pattern matching form is generally preferred.

Definition of Merge

Here is a definition of the `merge` function:

```
def merge(xs: List[Int], ys: List[Int]) = (xs, ys) match
  case (Nil, ys) => ys
  case (xs, Nil) => xs
  case (x :: xs1, y :: ys1) =>
    if x < y then x :: merge(xs1, ys)
    else y :: merge(xs, ys1)
```

Making Sort More General

Problem: How to parameterize `msort` so that it can also be used for lists with elements other than `Int`?

```
def msort[T](xs: List[T]): List[T] = ???
```

does not work, because the comparison < in `merge` is not defined for arbitrary types `T`.

Idea: Parameterize `merge` with the necessary comparison function.

Parameterization of Sort

The most flexible design is to make the function `sort` polymorphic and to pass the comparison operation as an additional parameter:

```
def msort[T](xs: List[T])(lt: (T, T) => Boolean) =
  ...
    merge(msort(fst)(lt), msort(snd)(lt))
```

Merge then needs to be adapted as follows:

```
def merge[T](xs: List[T], ys: List[T]) = (xs, ys) match
  ...
  case (x :: xs1, y :: ys1) =>
    if lt(x, y) then ...
    else ...
```

Calling Parameterized Sort

We can now call `msort` as follows:

```
val xs = List(-5, 6, 3, 2, 7)
val fruits = List("apple", "pear", "orange", "pineapple")

msort(xs)((x: Int, y: Int) => x < y)
msort(fruits)((x: String, y: String) => x.compareTo(y) < 0)
```

Or, since parameter types can be inferred from the call `msort(xs)`:

```
msort(xs)((x, y) => x < y)
```

Higher-Order List Functions

Principles of Functional Programming

Recurring Patterns for Computations on Lists

The examples have shown that functions on lists often have similar structures.

We can identify several recurring patterns, like,

- transforming each element in a list in a certain way,
- retrieving a list of all elements satisfying a criterion,
- combining the elements of a list using an operator.

Functional languages allow programmers to write generic functions that implement patterns such as these using higher-order functions.

Applying a Function to Elements of a List

A common operation is to transform each element of a list and then return the list of results.

For example, to multiply each element of a list by the same factor, you could write:

```
def scaleList(xs: List[Double], factor: Double): List[Double] = xs match
  case Nil     => xs
  case y :: ys => y * factor :: scaleList(ys, factor)
```

Mapping

This scheme can be generalized to the method `map` of the `List` class. A simple way to define `map` is as follows:

```
extension [T](xs: List[T])
  def map[U](f: T => U): List[U] = xs match
    case Nil      => xs
    case x :: xs => f(x) :: xs.map(f)
```

(in fact, the actual definition of `map` is a bit more complicated, because it is tail-recursive, and also because it works for arbitrary collections, not just lists).

Using `map`, `scaleList` can be written more concisely.

```
def scaleList(xs: List[Double], factor: Double) =
  xs.map(x => x * factor)
```

Exercise

Consider a function to square each element of a list, and return the result. Complete the two following equivalent definitions of `squareList`.

```
def squareList(xs: List[Int]): List[Int] = xs match
  case Nil      => ???
  case y :: ys => ???

def squareList(xs: List[Int]): List[Int] =
  xs.map(???)
```

Exercise

Consider a function to square each element of a list, and return the result. Complete the two following equivalent definitions of `squareList`.

```
def squareList(xs: List[Int]): List[Int] = xs match
  case Nil      => Nil
  case y :: ys => y * y :: squareList(ys)

def squareList(xs: List[Int]): List[Int] =
  xs.map(x => x * x)
```

Filtering

Another common operation on lists is the selection of all elements satisfying a given condition. For example:

```
def posElems(xs: List[Int]): List[Int] = xs match
  case Nil      => xs
  case y :: ys => if y > 0 then y :: posElems(ys) else posElems(ys)
```

Filter

This pattern is generalized by the method `filter` of the `List` class:

```
extension [T](xs: List[T])
  def filter(p: T => Boolean): List[T] = this match
    case Nil      => this
    case x :: xs => if p(x) then x :: xs.filter(p) else xs.filter(p)
```

Using `filter`, `posElems` can be written more concisely.

```
def posElems(xs: List[Int]): List[Int] =
  xs.filter(x => x > 0)
```

Variations of Filter

Besides filter, there are also the following methods that extract sublists based on a predicate:

`xs.filterNot(p)`	Same as `xs.filter(x => !p(x))`; The list consisting of those elements of `xs` that do not satisfy the predicate `p`.
`xs.partition(p)`	Same as `(xs.filter(p), xs.filterNot(p))`, but computed in a single traversal of the list `xs`.
`xs.takeWhile(p)`	The longest prefix of list `xs` consisting of elements that all satisfy the predicate `p`.
`xs.dropWhile(p)`	The remainder of the list `xs` after any leading elements satisfying `p` have been removed.
`xs.span(p)`	Same as `(xs.takeWhile(p), xs.dropWhile(p))` but computed in a single traversal of the list `xs`.

Exercise

Write a function pack that packs consecutive duplicates of list elements into sublists. For instance,

```
pack(List("a", "a", "a", "b", "c", "c", "a"))
```

should give

```
List(List("a", "a", "a"), List("b"), List("c", "c"), List("a")).
```

You can use the following template:

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

Exercise

Write a function pack that packs consecutive duplicates of list elements into sublists. For instance,

```
pack(List("a", "a", "a", "b", "c", "c", "a"))
```

should give

```
List(List("a", "a", "a"), List("b"), List("c", "c"), List("a")).
```

You can use the following template:

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

Exercise

Using `pack`, write a function `encode` that produces the run-length encoding of a list.

The idea is to encode `n` consecutive duplicates of an element `x` as a pair `(x, n)`. For instance,

```
encode(List("a", "a", "a", "b", "c", "c", "a"))
```

should give

```
List(("a", 3), ("b", 1), ("c", 2), ("a", 1)).
```

Exercise

Using pack, write a function encode that produces the run-length encoding of a list.

```
def encode[T](xs: List[T]): List[(T, Int)] = ???
```

Reduction of Lists

Principles of Functional Programming

Reduction of Lists

Another common operation on lists is to combine the elements of a list using a given operator.

For example:

```
sum(List(x1, ..., xn))          =  0 + x1 + ... + xn
product(List(x1, ..., xn))      =  1 * x1 * ... * xn
```

We can implement this with the usual recursive schema:

```
def sum(xs: List[Int]): Int = xs match
  case Nil      => 0
  case y :: ys => y + sum(ys)
```

ReduceLeft

This pattern can be abstracted out using the generic method `reduceLeft`:

`reduceLeft` inserts a given binary operator between adjacent elements of a list:

```
List(x1, ..., xn).reduceLeft(op)   =  x1.op(x2). ... .op(xn)
```

Using `reduceLeft`, we can simplify:

```
def sum(xs: List[Int])     = (0 :: xs).reduceLeft((x, y) => x + y)
def product(xs: List[Int]) = (1 :: xs).reduceLeft((x, y) => x * y)
```

A Shorter Way to Write Functions

Instead of `((x, y) => x * y))`, one can also write shorter:

```
(_ * _)
```

Every _ represents a new parameter, going from left to right.

The parameters are defined at the next outer pair of parentheses (or the whole expression if there are no enclosing parentheses).

So, `sum` and `product` can also be expressed like this:

```
def sum(xs: List[Int])     = (0 :: xs).reduceLeft(_ + _)
def product(xs: List[Int]) = (1 :: xs).reduceLeft(_ * _)
```

FoldLeft

The function `reduceLeft` is defined in terms of a more general function, `foldLeft`.

`foldLeft` is like `reduceLeft` but takes an *accumulator*, `z`, as an additional parameter, which is returned when `foldLeft` is called on an empty list.

```
List(x1, ..., xn).foldLeft(z)(op)   =  z.op(x1).op ... .op(xn)
```

So, `sum` and `product` can also be defined as follows:

```
def sum(xs: List[Int])      =  xs.foldLeft(0)(_ + _)
def product(xs: List[Int])  =  xs.foldLeft(1)(_ * _)
```

Implementations of ReduceLeft and FoldLeft

`foldLeft` and `reduceLeft` can be implemented in class `List` as follows.

```
abstract class List[T]:

  def reduceLeft(op: (T, T) => T): T = this match
    case Nil     => throw IllegalOperationException("Nil.reduceLeft")
    case x :: xs => xs.foldLeft(x)(op)

  def foldLeft[U](z: U)(op: (U, T) => U): U = this match
    case Nil     => z
    case x :: xs => xs.foldLeft(op(z, x))(op)
```

FoldRight and ReduceRight

Applications of `foldLeft` and `reduceLeft` unfold on trees that lean to the left.

They have two dual functions, `foldRight` and `reduceRight`, which produce trees which lean to the right, i.e.,

```
List(x1, ..., x{n-1}, xn).reduceRight(op) = x1.op(x2.op( ... x{n-1}.op(xn) ...
List(x1, ..., xn).foldRight(z)(op )       = x1.op(x2.op( ... xn.op(z) ...))
```

Implementation of FoldRight and ReduceRight

They are defined as follows

```
def reduceRight(op: (T, T) => T): T = this match
  case Nil      => throw UnsupportedOperationException("Nil.reduceRight")
  case x :: Nil => x
  case x :: xs  => op(x, xs.reduceRight(op))

def foldRight[U](z: U)(op: (T, U) => U): U = this match
  case Nil     => z
  case x :: xs => op(x, xs.foldRight(z)(op))
```

Difference between FoldLeft and FoldRight

For operators that are associative and commutative, `foldLeft` and `foldRight` are equivalent (even though there may be a difference in efficiency).

But sometimes, only one of the two operators is appropriate.

Exercise

Here is another formulation of `concat`:

```
def concat[T](xs: List[T], ys: List[T]): List[T] =
  xs.foldRight(ys)(_ :: _)
```

Here, it isn't possible to replace `foldRight` by `foldLeft`. Why?

```
O      The types would not work out
O      The resulting function would not terminate
O      The result would be reversed
```

Exercise

Here is another formulation of `concat`:

```
def concat[T](xs: List[T], ys: List[T]): List[T] =
  xs.foldRight(ys)(_ :: _)
```

Here, it isn't possible to replace `foldRight` by `foldLeft`. Why?

```
X      The types would not work out
O      The resulting function would not terminate
O      The result would be reversed
```

Back to Reversing Lists

We now develop a function for reversing lists which has a linear cost.

The idea is to use the operation `foldLeft`:

```
def reverse[T](xs: List[T]): List[T] = xs.foldLeft(z?)(op?)
```

All that remains is to replace the parts `z?` and `op?`.

Let's try to *compute* them from examples.

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
Nil
```

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
Nil

=   reverse(Nil)
```

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
Nil

=   reverse(Nil)

=   Nil.foldLeft(z?)(op)
```

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
  Nil

=   reverse(Nil)

=   Nil.foldLeft(z?)(op)

=   z?
```

Consequently, `z? = Nil`

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)
```

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)

=   reverse(List(x))
```

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)

=   reverse(List(x))

=   List(x).foldLeft(Nil)(op?)
```

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)

=   reverse(List(x))

=   List(x).foldLeft(Nil)(op?)

=   op?(Nil, x)
```

Consequently, `op?(Nil, x) = List(x) = x :: Nil`.

This suggests to take for `op?` the operator `::` but with its operands swapped.

Deduction of Reverse(3)

We thus arrive at the following implementation of `reverse`.

```
def reverse[a](xs: List[T]): List[T] =
  xs.foldLeft(List[T]())((xs, x) => x :: xs)
```

Remark: the type parameter in `List[T]()` is necessary for type inference.

Q: What is the complexity of this implementation of `reverse` ?

Deduction of Reverse(3)

We thus arrive at the following implementation of `reverse`.

```
def reverse[a](xs: List[T]): List[T] =
  xs.foldLeft(List[T]())((xs, x) => x :: xs)
```

Remark: the type parameter in `List[T]()` is necessary for type inference.

Q: What is the complexity of this implementation of `reverse` ?

A: Linear in `xs`

Exercise

Complete the following definitions of the basic functions `map` and `length` on lists, such that their implementation uses `foldRight`:

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  xs.foldRight(List[U]())( ??? )

def lengthFun[T](xs: List[T]): Int =
  xs.foldRight(0)( ??? )
```

Exercise

Complete the following definitions of the basic functions `map` and `length` on lists, such that their implementation uses `foldRight`:

```scala
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  xs.foldRight(List[U]())((y, ys) => f(y) :: ys)

def lengthFun[T](xs: List[T]): Int =
  xs.foldRight(0)((y, n) => n + 1)
```

Reasoning About Lists

Principles of Functional Programming

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

$\boxed{\textbf{Base case: } 4}$

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

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

For the left-hand side we have:

```
(Nil ++ ys) ++ zs

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

Base Case

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

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

$\boxed{\textbf{Induction step: } \texttt{x :: xs}}$

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

For the right hand side we have:

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

Induction Step: RHS

For the right hand side we have:

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

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

So this case (and with it, the property) is established.

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
O	3
O	4