Week 5 : More on Lists

Reduction of Lists

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

For example:

$$\begin{aligned} sum(List(x_1, ..., x_n)) &= 0 + x_1 + ... + x_n \\ product(List(x_1, ..., x_n)) &= 1 * x_1 * ... * x_n \end{aligned}$$

We can implement this by using the usual recursive scheme:

```
def sum(xs: List[Int]): Int = xs match {
  case Nil ⇒ 0
  case y :: ys ⇒ y + sum(ys)
}
def product(xs: List[Int]): Int = xs match {
  case Nil ⇒ 1
  case y :: ys ⇒ y * product(ys)
}
```


The generic method *reduceLeft* inserts a given binary operator between two adjacent elements.

For example.

$$List(x_1, ..., x_n).reduceLeft(op) = (...(x_1\ op\ x_2)\ op\ ...\ )\ op\ x_n$$

It's now possible to write more simply:

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


Implementation of reduceLeft

How can we implement *reduceLeft*?

```
abstract class List[a] { ...
  def reduceLeft(op: (a, a) ⇒ a): a = this match {
    case Nil ⇒ error("Nil.reduceLeft")
    case x :: xs ⇒ (xs foldLeft x)(op)
  }
  def foldLeft[b](z: b)(op: (b, a) ⇒ b): b = this match {
    case Nil ⇒ z
    case x :: xs ⇒ (xs foldLeft op(z, x))(op)
  }
}
```

The function *reduceLeft* is defined in terms of another function which is often useful, *foldLeft*.

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

In other words,

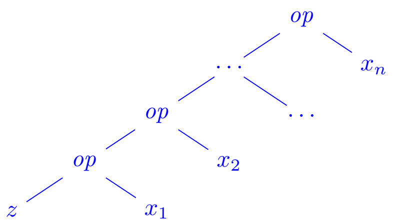
$$(List(x_1, ..., x_n)\ foldLeft\ z)(op) \quad = \quad (...(z\ op\ x_1)\ op\ ...\ )\ op\ x_n$$

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

$$\begin{aligned} &\textbf{\textit{def}}\ sum(xs\colon List[Int]) &&= (xs\ foldLeft\ 0)\ \{(x, y) \Rightarrow x + y\} \\ &\textbf{\textit{def}}\ product(xs\colon List[Int]) &&= (xs\ foldLeft\ 1)\ \{(x, y) \Rightarrow x * y\} \end{aligned}$$

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.,

$$\begin{aligned} &List(x_1, ..., x_n).reduceRight(op) &&= x_1\ op\ (\ ...\ (x_{n-1}\ op\ x_n)...) \\ &(List(x_1, ..., x_n)\ foldRight\ acc)(op) &&= x_1\ op\ (\ ...\ (x_n\ op\ acc)...) \end{aligned}$$

They are defined as follows


```
def reduceRight(op: (a, a) ⇒ a): a = this match {
    case Nil ⇒ error("Nil.reduceRight")
    case x :: Nil ⇒ x
    case x :: xs ⇒ op(x, xs.reduceRight(op))
}
def foldRight[b](z: b)(op: (a, b) ⇒ b): b = this match {
    case Nil ⇒ z
    case x :: xs ⇒ op(x, (xs foldRight z)(op))
}
```

For operators that are both 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.

Example: Here is another formulation of *concat*:

```
def concat[a](xs: List[a], ys: List[a]): List[a] =
    (xs foldRight ys) {(x, xs) ⇒ x :: xs}
```

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

Back to Reversing Lists

Here is a function for reversing lists which has a linear cost.

The idea is to use the operation *foldLeft*:

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

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

Let's try to deduce them from examples.

To start,

Base Case: *List()*

	reverse(List())	*(by specification of reverse)*
=	*(List() foldLeft z)(op)*	*(by definition of reverse)*
=	*z*	*(by definition of foldLeft)*

Consequently, $z = List()$.

Then,

Induction Step: $\boxed{\textit{List}(x)}$

$$\begin{aligned} & \textit{reverse}(\textit{List}(x)) && (\text{by specification } \textit{reverse}) \\ = \; & (\textit{List}(x) \textit{ foldLeft List}())(\textit{op}) && (\text{by def. of } \textit{reverse} \text{ with } z = \textit{List}()) \\ = \; & \textit{op}(\textit{List}(), x) && (\text{by definition of } \textit{foldLeft}) \end{aligned}$$

Consequently, $op(List(), x) = List(x) = x :: List()$. This suggests to take for *op* the operator :: and swapping its operands.

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

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

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

Q: What's the complexity of this implementation of *reverse* ?

More on Fold and Reduce

Exercise: Complete the following definitions, based on the usage of *foldRight*, which introduce base operations for manipulating lists.

```
def mapFun[a, b](xs: List[a], f: a => b): List[b] =
  (xs foldRight List[b]()){ ?? }

def lengthFun[a](xs: List[a]): Int =
  (xs foldRight 0){ ?? }
```


Handling Nested Lists

We can extend the usage of higher order functions on lists to many calculations which are usually expressed using nested loops.

Example: Given a positive integer n, find all pairs of positive integers i and j, with $1 \leq j < i < n$ such that $i + j$ is prime.

For example, if $n = 7$, the sought pairs are

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i + j$	3	5	5	7	7	7	11

A natural way to do this is to:

- Generate the sequence of all pairs of integers (i, j) such that $1 \leq j < i < n$.
- Filter the pairs for which $i + j$ is prime.

One natural way to generate the sequence of pairs is to:

- Generate all the integers i between 1 and n (excluded). This can be realized by the function

  ```
  def range(from: Int, end: Int): List[Int] =
    if (from >= end) List()
    else from :: range(from + 1, end)
  ```

  which is predefined in *List*.
- For each integer i, generate the list of pairs $(i, 1), \ldots, (i, i-1)$. This can be achieved by combining *range* and *map*:

  ```
  List.range(1, i) map (x => (i, x))
  ```
- Finally, combine all the sub-lists using *foldRight* with :::.

By reassembling the pieces, we obtain the following expression:

```
List.range(1, n)
  .map(i ⇒ List.range(1, i).map(x ⇒ (i, x)))
  .foldRight(List[(Int, Int)]()) {(xs, ys) ⇒ xs ::: ys}
  .filter(pair ⇒ isPrime(pair._1 + pair._2))
```


The *flatMap* Function

The combination of applying a function to the elements of a list and then concatenating the results is so common, that we have introduced a special method for this in *List.scala*:

```
abstract class List[a] { ...
  def flatMap[b](f: a ⇒ List[b]): List[b] = this match {
    case Nil ⇒ Nil
    case x :: xs ⇒ f(x) ::: (xs flatMap f)
  }
}
```

With *flatMap*, we could have written an expression more concisely:

```
List.range(1, n)
  .flatMap(i ⇒ List.range(1, i).map(x ⇒ (i, x)))
  .filter(pair ⇒ isPrime(pair._1 + pair._2))
```

Q: Find a concise way to define *isPrime*. (Hint: Use *forall* defined in *List*).

The *zip* Function

The *zip* method in the *List* class combines two lists into one list of pairs.

```
abstract class List[a] { ...
  def zip[b](that: List[b]): List[(a,b)] =
    if (this.isEmpty || that.isEmpty) Nil
    else (this.head, that.head) :: (this.tail zip that.tail)
```

Example: By using *zip* and *foldLeft*, we can define the scalar product of two lists in the following way.

```
def scalarProduct(xs: List[Double], ys: List[Double]): Double =
  (xs zip ys)
  .map(xy ⇒ xy._1 * xy._2)
  .foldLeft(0.0){(x, y) ⇒ x + y}
```


Summary

- We have seen that lists are a fundamental data structure in functional programming.
- Lists are defined by parametric classes and are manipulated by polymorphic methods.
- Lists are in functional languages what arrays are in imperative languages.
- But contrary to arrays, we normally don't access elements of a list using their index.
- We prefer to traverse lists recursively or via higher-order combinators such as *map*, *filter*, *foldLeft* or *foldRight*.

Reasoning About Lists

Recall the concatenation operation on lists (seen during week 4)

```
class List[a] {
  ...
  def ::: (that: List[a]): List[a] = that match {
    case Nil => this
    case x :: xs => x :: (xs ::: this)
  }
}
```

We would like to verify that the concatenation is associative, and that it admits the empty list *List*() as neutral element to the left and to the right:

$$(xs ::: ys) ::: zs \quad = \quad xs ::: (ys ::: zs)$$

$$xs ::: List() \quad = \quad xs \quad = \quad List() ::: xs$$

Q: How can we prove properties like these?

A: By structural induction on lists.

Reminder: Natural Induction (or Recurrence)

Recall the principle of proof by natural induction:

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

1. Show that we have $P(b)$ (base case),
2. for all integers $n \geq b$ show that:

   if one has $P(n)$, then one also has $P(n+1)$

   (induction step).

Example: Given

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

Show that, for all $n \geq 4$,

$$factorial(n) \geq 2^n$$

Base Case: 4

This case is established by simple calculations of $factorial(4) = 24$ and $2^4 = 16$.

Induction Step: n+1 We have for $n \geq 4$:

$$\begin{array}{lll} & factorial(n+1) & \\ = & (n+1) * factorial(n) & (\textit{by the 2nd clause of factorial (*)}) \\ \geq & 2 * factorial(n) & (\textit{by calculating}) \\ \geq & 2 * 2^n. & (\textit{by induction hypothesis}) \end{array}$$

Note that a proof can freely apply reduction steps like (*) to the interior 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:

In the case of lists, it has the following form:

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

1. show that $P(List())$ holds (base case),
2. for a list xs and some element x, show that:

   if $P(xs)$ holds, then $P(x :: xs)$ also holds

   (induction step).

Example

We will show that $(xs ::: ys) ::: zs = xs ::: (ys ::: zs)$, by structural induction on xs.

Base Case: $List()$

For the left-hand side we have:

$$\begin{array}{lll} & (List() ::: ys) ::: zs & \\ = & ys ::: zs & (\textit{by the first clause of} :::) \end{array}$$

For the right-hand side, we have:

$$\begin{array}{lll} & List() ::: (ys ::: zs) & \\ = & ys ::: zs & (\textit{by the first clause of} :::) \end{array}$$

This case is therefore established.

Induction Step: $x :: xs$

For the left-hand side, we have:

$$\begin{array}{lll} & ((x :: xs) ::: ys) ::: zs & \\ = & (x :: (xs ::: ys)) ::: zs & (\textit{by the second clause of} :::) \\ = & x :: ((xs ::: ys) ::: zs) & (\textit{by the second clause of} :::) \\ = & x :: (xs ::: (ys ::: zs)) & (\textit{by induction hypothesis}) \end{array}$$

For the right hand side we have:

$$\begin{array}{lll} & (x :: xs) ::: (ys ::: zs) & \\ = & x :: (xs ::: (ys ::: zs)) & (\textit{by the second clause of} :::) \end{array}$$

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

Exercise: Show by induction on xs that $xs ::: List() = xs$.

Example (2)

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

```
abstract class List[a] { ...
  def reverse: List[a] = this match {
    case List() => List()                       /* 1st clause */
    case x :: xs => xs.reverse ::: List(x)      /* 2nd clause */
  }
}
```

We'd like to prove the following proposition

$$xs.reverse.reverse = xs$$

We proceed by induction on xs. The base case is easy to establish:

$$\begin{array}{lll} & List().reverse.reverse & \\ = & List().reverse & (\textit{by the 1st clause of } reverse) \\ = & List() & (\textit{by the 1st clause of } reverse) \end{array}$$

For the induction step, we try:

$$\begin{array}{lll} & (x :: xs).reverse.reverse & \\ = & (xs.reverse ::: List(x)).reverse & (\textit{by the 2nd clause of } reverse) \end{array}$$

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

$$\begin{array}{lll} & x :: xs & \\ = & x :: xs.reverse.reverse & (\textit{by induction}) \end{array}$$

Both sides are simplified in different expressions.

We must still show that

$$(xs.reverse ::: List(x)).reverse = x :: xs.reverse.reverse$$

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

We must instead try to *generalize* the equation:

$$(ys ::: List(x)).reverse = x :: ys.reverse$$

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

Exercise: Is it true that $(xs\ drop\ m)\ apply\ n = xs\ apply\ (m + n)$ for all integers $m \geq 0$, $n \geq 0$ and all lists *xs* ?

Structural Induction on Trees

Structural induction is not limited to lists; it applies to any tree structure.

The general induction principle is the following:

To show the property $P(t)$ for all trees of a certain type,

- show $P(l)$ for all the leaves l of the tree,
- for each internal node t with sub-trees s_1, ..., s_n, show that $P(s_1) \wedge ... \wedge P(s_n) \Rightarrow P(t)$.

Example: Recall our definition of *IntSet* with the operations *contains* and *incl*:

```
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}
```



```
case class Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
}
case class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: Int): IntSet =
    if (x < elem) NonEmpty(elem, left incl x, right)
    else if (x > elem) NonEmpty(elem, left, right incl x)
    else this
}
```

(With ***case*** modifiers to enable the use of factory methods in place of ***new***).

What does it mean to prove the correctness of this implementation?

The Laws of IntSet

One way to define and show the correctness of an implementation consists of proving the laws that it respects.

In the case of *IntSet*, we have the following three laws:

For any set s, and elements x and y:

Empty contains x	=	***false***	
(s incl x) contains x	=	***true***	
(s incl x) contains y	=	*s contains y*	***if*** $x \neq y$

(In fact, we can show that these laws completely characterize the desired data type).

How can we prove these laws?

Proposition 1: *Empty contains x* = ***false***.

Proof: According to the definition of *contains* in *Empty*.

Proposition 2: $(s\ incl\ x)\ contains\ x = \textbf{true}$

Proof:

Base Case: *Empty*

$$
\begin{aligned}
& (Empty\ incl\ x)\ contains\ x \\
= \quad & \text{(by the definition of } incl \text{ in } Empty\text{)} \\
& NonEmpty(x, Empty, Empty)\ contains\ x \\
= \quad & \text{(by the definition of } contains \text{ in } NonEmpty\text{)} \\
& \textbf{true}
\end{aligned}
$$

Induction Step: $NonEmpty(x, l, r)$

$$
\begin{aligned}
& (NonEmpty(x, l, r)\ incl\ x)\ contains\ x \\
= \quad & \text{(by the definition of } incl \text{ in } NonEmpty\text{)} \\
& NonEmpty(x, l, r)\ contains\ x \\
= \quad & \text{(by the definition of } contains \text{ in } NonEmpty\text{)} \\
& \textbf{true}
\end{aligned}
$$

Induction Step: $NonEmpty(y, l, r)$ with $y < x$

$$
\begin{aligned}
& (NonEmpty(y, l, r)\ incl\ x)\ contains\ x \\
= \quad & \text{(by the definition of } incl \text{ in } NonEmpty\text{)} \\
& NonEmpty(y, l, r\ incl\ x)\ contains\ x \\
= \quad & \text{(by the definition of } contains \text{ in } NonEmpty\text{)} \\
& (r\ incl\ x)\ contains\ x \\
= \quad & \text{(by the induction hypothesis)} \\
& \textbf{true}
\end{aligned}
$$

Induction Step: $NonEmpty(y, l, r)$ with $y > x$ is analogous.

Proposition 3: If $x \neq y$ then $xs\ incl\ y\ contains\ x \ = \ xs\ contains\ x$.

Proof: See blackboard.

Exercise

Suppose we add a function *union* to *IntSet*:

```
abstract class IntSet { ...
   def union(other: IntSet): IntSet
}
class Expty extends IntSet { ...
   def union(other: IntSet) = other
}
class NonEmpty(x: Int, l: IntSet, r: IntSet) extends IntSet { ...
   def union(other: IntSet): IntSet = l union (r union (other incl x))
}
```

The correctness of *union* can be translated into the following law:

Proposition 4: $(xs\ union\ ys)\ contains\ x \ = \ xs\ contains\ x\ ||\ ys\ contains\ x$.
Is this true? Which hypothesis is missing? Find a counter-example.

Show proposition 4 by using structural induction on xs.