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:

\[
\text{sum}(\text{List}(x_1, \ldots, x_n)) = 0 + x_1 + \ldots + x_n
\]

\[
\text{product}(\text{List}(x_1, \ldots, x_n)) = 1 \times x_1 \times \ldots \times x_n
\]

We can implement this by using the usual recursive scheme:

```python
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 \texttt{reduceLeft} inserts a given binary operator between two adjacent elements.

For example,

\[
\text{List}(x_1, \ldots, x_n).\text{reduceLeft}(op) = (\ldots(x_1 op x_2) op \ldots) op x_n
\]

It's now possible to write more simply:

```python
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)
```

The function \texttt{reduceLeft} is defined in terms of another function which is often useful, \texttt{foldLeft}.

Implementation of reduceLeft

How can we implement \texttt{reduceLeft}?

```scala
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](op : (a, b) ⇒ b) : (a, b) = this match {
    case Nil ⇒ (x, x)
    case x :: xs ⇒ (xs foldLeft op(x, x))(op)
  }
}
```

The function \texttt{reduceLeft} is defined in terms of another function which is often useful, \texttt{foldLeft}.
**FoldRight and ReduceRight**

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

\[
\begin{align*}
\text{sum} &= \text{foldLeft} (\text{List}[\text{Int}]) = (\text{List}[\text{Int}]) \{ (x, y) \Rightarrow x + y \} \\
\text{product} &= \text{foldLeft} (\text{List}[\text{Int}]) = (\text{List}[\text{Int}]) \{ (x, y) \Rightarrow x \times y \}
\end{align*}
\]

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

\[
\begin{align*}
\text{List}(x_1, \ldots, x_n).\text{reduceRight}(\text{op}) &= x_1 \text{ op } (\ldots (x_{n-1} \text{ op } x_n)\ldots) \\
(\text{List}(x_1, \ldots, x_n).\text{foldRight}(\text{acc}))(\text{op}) &= x_1 \text{ op } (\ldots (x_n \text{ op } \text{acc})\ldots)
\end{align*}
\]

They are defined as follows:

\[
\begin{align*}
\text{def } \text{reduceRight}(\text{op}: (a, a) \Rightarrow a): a = \text{this match } \{ \\
\text{case } \text{Nil} \Rightarrow \text{error("Nil.reduceRight") } \\
\text{case } x :: \text{Nil} \Rightarrow x \\
\text{case } x :: \text{xs} \Rightarrow \text{op}(x, x :: \text{reduceRight}(\text{op})) \\
\} \\
\text{def } \text{foldRight}(\text{acc}, b): b = \text{this match } \{ \\
\text{case } \text{Nil} \Rightarrow z \\
\text{case } x :: \text{xs} \Rightarrow \text{op}(x, x :: \text{foldRight}(\text{acc}, z))(\text{op}) \\
\}
\]

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:**

\[
\begin{align*}
\text{def } \text{concat}(a): \text{List}[a], \text{ys}: \text{List}[a]): \text{List}[a] = \\
(\text{xs foldRight } \text{ys}) \{ (x, x :: \text{xs} ) \Rightarrow x :: \text{xs} \}
\end{align*}
\]

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:

\[
\text{def } \text{reverse}[: a] (\text{xs}: \text{List}[a]): \text{List}[a] = (\text{xs foldLeft } z')(\text{op})
\]

All that remains is to replace the parts \(z'\) and \(\text{op'}\).

Let's try to deduce them from examples.

To start,

**Base Case:** \(\text{List}()\)

\[
\begin{align*}
\text{reverse}(\text{List}()) &= (\text{List}() \text{ foldLeft } z')(\text{op}) \\
&= z
\end{align*}
\]

Consequently, \(z = \text{List}()\).
Then,

**Induction Step:** \( \text{List}(x) \)

\[
\text{reverse}(\text{List}(x)) = (\text{List}(x) \text{foldLeft} \text{List}())(\text{op})
\]

(by def. of reverse with \( z = \text{List}() \))

(by def. of \text{foldLeft})

Consequently, \( \text{op}(\text{List}(), x) = \text{List}() \) (by def. of \text{foldLeft})

Remark: the type parameter in \( \text{List}[a]() \) is necessary for type inference.

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

### 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 | 5 | 6 | 6 |
| \( j \) | 1 | 2/1 | 3 | 2 | 1 | 5 |
| \( i + j \) | 3 | 5 | 5 | 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:

\[
\text{def} \text{range(from : Int, end : Int): List[Int]} =
\]

\[
\text{if}(\text{from} \geq \text{end}) \text{List}()
\]

\[
\text{else from :: range(from + 1, end)}
\]

which is predefined in \text{List}.

- For each integer \( i \), generate the list of pairs \( (i, 1), \ldots, (i, i-1) \). This can be achieved by combining \text{range} and \text{map}:

\[
\text{List.range(1, i) map (x ⇒ (i, x))}
\]

- Finally, combine all the sub-lists using \text{foldRight} with \( :: \).

### More on Fold and Reduce

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

\[
\text{def mapFun[a, b](xs: List[a], f: a ⇒ b): List[b]} =
\]

\[
(\text{xs foldRight List[b]()})\{ ?? \}
\]

\[
\text{def lengthFun[a](xs : List[a]): Int} =
\]

\[
(\text{xs foldRight 0})\{ ?? \}
\]
By reassembling the pieces, we obtain the following expression:

\[
\text{List.range}(1, n) \map{i}{\text{List.range}(1, i)} \map{x}{(i, x)} \text{foldRight(List[(Int, Int)])} \left\{ (xs, ys) \Rightarrow xs :: ys \right\} \text{filter}(\text{pair} \Rightarrow \text{isPrime}(\text{pair}_1 + \text{pair}_2))
\]

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:

```scala
abstract class List[a] { ...
  def flatMap[(b) => List[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:

\[
\text{List.range}(1, n) \map{i}{\text{List.range}(1, i)} \map{x}{(i, x)} \text{filter}(\text{pair} \Rightarrow \text{isPrime}(\text{pair}_1 + \text{pair}_2))
\]

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.

```scala
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.

```scala
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:

\[(xs :::: ys) :::: zs = xs :::: (ys :::: zs)\]
\[xs :::: List() = xs = List() :::: xs\]

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 \(n \geq 4\):

\[factorial(n + 1) = (n + 1) * factorial(n) \quad \text{(by the 2nd clause of factorial (*))}\]
\[\geq 2 * factorial(n) \quad \text{(by calculating)}\]
\[\geq 2 * 2^n \quad \text{(by induction hypothesis)}\]

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: \(\text{List}()\)**

For the left-hand side we have:

\[
(\text{List}() :: ys) :: zs = ys :: zs \quad (\text{by the first clause of} ::)
\]

For the right-hand side, we have:

\[
\text{List}() :: (ys :: zs) = ys :: zs \quad (\text{by the first clause of} ::)
\]

This case is therefore established.

Induction Step: \(x :: xs\)

For the left-hand side, we have:

\[
((x :: xs) :: ys) :: zs = (x :: (xs :: ys)) :: zs \quad (\text{by the second clause of} ::)
\]

For the right-hand side we have:

\[
x :: (xs :: (ys :: zs)) = x :: (xs :: (ys :: zs)) \quad (\text{by the second clause of} ::)
\]

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

**Example (2)**

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

```scala
abstract class List[a] {...
def reverse: List[a] = this match {
  case List() => List()
  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:

\[
\text{List}().reverse.reverse = \text{List}() \quad (\text{by the 1st clause of reverse})
\]

For the induction step, we try:

\[
(x :: xs).reverse.reverse
\]

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

\[
x :: xs
\]

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 $(xs \text{ drop } m) \text{ apply } n = xs \text{ apply } (m + n)$ 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 $s_1, \ldots, s_n$, show that $P(s_1) \land \ldots \land P(s_n) \Rightarrow P(t)$.

**Example:** Recall our definition of $\text{IntSet}$ with the operations $\text{contains}$ and $\text{incl}$:

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

```scala
case class Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet =...
}
```

```scala
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 $\text{IntSet}$, we have the following three laws:

For any set $s$, and elements $x$ and $y$:
- $\text{Empty contains } x = false$
- $(s \text{ incl } x) \text{ contains } x = true$
- $(s \text{ incl } x) \text{ contains } y = s \text{ contains } y$ 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:** $\text{Empty contains } x = false$.

**Proof:** According to the definition of $\text{contains}$ in $\text{Empty}$.
Proposition 2: \((s \text{ incl } x) \text{ contains } x = \text{true}\)

Proof:

Base Case: Empty
\((\text{Empty incl } x) \text{ contains } x = \text{true}\) by the definition of incl in Empty

Induction Step: NonEmpty(x, l, r)
\((\text{NonEmpty}(x, l, r) \text{ incl } x) \text{ contains } x = \text{true}\) by the definition of contains in NonEmpty

Induction Step: NonEmpty(y, l, r) with \(y < x\)
\((\text{NonEmpty}(y, l, r) \text{ incl } x) \text{ contains } x = \text{true}\) by the definition of incl in NonEmpty

Proposition 3: If \(x \neq y\) then \(xs \text{ incl } y \text{ contains } x = xs \text{ contains } x\).

Proof: See blackboard.

Exercise
Suppose we add a function union to IntSet:

\begin{verbatim}
abstract class IntSet {
    def union(other : IntSet): IntSet
} class Empty 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))
}
\end{verbatim}

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

Proposition 4: \((xs \text{ union } ys) \text{ contains } x = xs \text{ contains } x \cup ys \text{ contains } x\).

Is this true? Which hypothesis is missing? Find a counter-example.

Show proposition 4 by using structural induction on \(xs\).