Week 5: More On Lists

Reducing Lists

Another common operation is to combine the elements of a list with some operator.

For instance:

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

These can be implemented with 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 generalization *reduceLeft* inserts a given binary operator between adjacent elements.

E.g.

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

Then we can simply write:

$$\begin{array}{lcl} \textbf{def}\; sum\,(xs\colon List\,[int\,]) & = & (0 :: xs)\; reduceLeft\; \{\,(x, y) \Rightarrow x + y\} \\ \textbf{def}\; product\,(xs\colon List\,[int\,]) & = & (1 :: xs)\; reduceLeft\; \{\,(x, y) \Rightarrow x * y\} \end{array}$$

Implementation of ReduceLeft

How can *reduceLeft* be implemented?

```
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 *reduceLeft* function is defined in terms of another generally useful function, *foldLeft*.

foldLeft takes as additional parameter an *accumulator* z, which is returned for empty lists.

That is,

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

So *sum* and *product* could be defined alternatively as follows.

$$\begin{array}{lcl} \textbf{def}\ sum\,(xs: List\,[int]) & = & (xs\ foldLeft\ 0)\ \{(x, y) \Rightarrow x + y\} \\ \textbf{def}\ product\,(xs: List\,[int]) & = & (xs\ foldLeft\ 1)\ \{(x, y) \Rightarrow x * y\} \end{array}$$

FoldRight and ReduceRight

Applications of *foldLeft* and *reduceLeft* expand to left-leaning trees:

They have duals foldRight and reduceRight, which produce right-leaning trees. I.e.

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

These are defined as follows.

```
def reduceRight(op: (a, a) => a): a = 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 = match {
    case Nil => z
    case x :: xs => op(x, (xs foldRight z)(op))
}
```

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

But sometimes, only one of the two operators is appropriate or has the right type:

Example: Here is an alternative formulation of *concat*:

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

Here it is not possible to replace the *foldRight* with *foldLeft*. (Why not?)

List Reversal Again

Here is a list reversal function with linear cost.

The idea is to use a *foldLeft* operation:

$$\textbf{\textit{def}}\ \mathit{reverse}[a](\mathit{xs}: \mathit{List}[a]): \mathit{List}[a] = (\mathit{xs}\ \mathit{foldLeft}\ z?)(\mathit{op}?)$$

We only need to fill in the *z?* and *op?* parts.

Let's try to deduce them from examples.

First,

$$\begin{aligned} & \mathit{List}() \\ = \ & \mathit{reverse}(\mathit{List}()) && \textit{// by specification} \\ = \ & (\mathit{List}()\ \mathit{foldLeft}\ z)(\mathit{op}) && \textit{// by the template for reverse} \\ = \ & z && \textit{// by definition of foldLeft} \end{aligned}$$

Hence, $z = \mathit{List}()$.

Second,

$$
\begin{aligned}
& \mathit{List}\,(x) \\
= \;& \mathit{reverse}\,(\mathit{List}\,(x)) && \textit{// by specification} \\
= \;& (\mathit{List}\,(x)\ \mathit{foldLeft}\ \mathit{List}\,())\,(\mathit{op}) && \textit{// by the template for reverse, with } z = \mathit{List}() \\
= \;& \mathit{op}\,(\mathit{List}\,(),\, x) && \textit{// by definition of foldLeft}
\end{aligned}
$$

Hence, $\mathit{op}\,(\mathit{List}\,(),\, x) = \mathit{List}\,(x) = x :: \mathit{List}\,()$. This suggests to take as *op* the :: operator with its operands exchanged.

Hence, we arrive at the following implementation for *reverse*.

$$
\begin{aligned}
& \textbf{\textit{def}}\ \mathit{reverse}\,[\mathrm{a}]\,(\mathit{xs}\colon \mathit{List}\,[\mathrm{a}])\colon \mathit{List}\,[\mathrm{a}] = \\
& \quad (\mathit{xs}\ \mathit{foldLeft}\ \mathit{List}\,[\mathrm{a}]\,())\{\,(\mathit{xs},\, x) \Rightarrow x :: \mathit{xs}\}
\end{aligned}
$$

Remark: The type parameter in $\mathit{List}\,[\mathrm{a}]\,()$ is necessary to make the type inferencer work.

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

More on Fold and Reduce

Exercise: Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as fold operations.

$\textbf{\textit{def}}\ \textit{mapFun}\,[a,\ b]\,(\textit{xs}: \textit{List}\,[a],\ f\colon a \Rightarrow b)\colon \textit{List}\,[b] =$
$\quad (\textit{xs foldRight List}\,[b]\,(\,))\{\ ??\ \}$

$\textbf{\textit{def}}\ \textit{lengthFun}\,[a]\,(\textit{xs}: \textit{List}\,[a])\colon \textit{int} =$
$\quad (\textit{xs foldRight } 0)\{\ ??\ \}$

Nested Mappings

We can extend the higher-order list functions to include many computations that are normally expressed as nested loops.

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

For example, if $n = 7$, the 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:

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

A natural way to generate the sequence of pairs is:

- Generate all integers between *1* and n (excluded) for i. This can be packaged using the function

```
def range(from: int, end: int): List[int] =
  if (from >= end) scala.Predef.List()
  else from :: range(from + 1, end);
```

  which is predefined in module *List*.

- For each integer i, generate the list of pairs $(i, 1)$, ..., $(i, i-1)$. This can be achieved by a combination of *range* and *map*:

```
List.range(1, i) map (x => Pair(i, x))
```

- Finally, combine all sublists using *foldRight* with :::.

Putting everything together gives the following expression:

$$
\begin{array}{l}
\mathit{List.range}\,(1,\ n\,) \\
\quad .\mathit{map}\,(i \Rightarrow \mathit{List.range}\,(1,\ i\,).\mathit{map}\,(x \Rightarrow \mathit{Pair}\,(i,\ x\,)\,)\,) \\
\quad .\mathit{foldRight}\,(\mathit{List}\,[\mathit{Pair}\,[\mathit{int},\ \mathit{int}\,]\,]\,(\,)\,)\ \{\,(\mathit{xs},\ \mathit{ys}\,) \Rightarrow \mathit{xs}\ :::\ \mathit{ys}\} \\
\quad .\mathit{filter}\,(\mathit{pair} \Rightarrow \mathit{isPrime}\,(\mathit{pair.\_1} + \mathit{pair.\_2}\,)\,)
\end{array}
$$

Function *flatMap*

The combination of mapping and then concatenating sublists resulting from the map is so common that we there is a special method for it in *List.scala*:

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

With *flatMap*, our expression could have been written more concisely as follows.

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

Q: What is a concise way to define *isPrime*? (Hint: use *forall* in *List*).

Function *zip*

The *zip* method in *List* combines two lists into a list of pairs.

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

Example: Using *zip* and *foldLeft*, we can define the scalar product of two lists as follows.

```
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 encountered the list as a fundamental data structure in functional programming.
- Lists are defined by parameterized classes, and operated upon by polymorphic methods.
- Lists are the analogue of arrays in imperative languages.
- But unlike arrays, lists elements are usually not accessed by their index.
- Instead, lists are traversed recurisvely or via higher-order combinators such as *map*, *filter*, *foldLeft*, *foldRight*.

Reasoning About Lists

Recall the concatenation operation for lists:

```
class List[a] {
  ...
  def ::: (that: List[a]): List[a] =
    if (isEmpty) that
    else head :: (tail ::: that)
}
```

We would like to verify that concatenation is associative, with the empty list *List*( ) as left and right identity:

$$
\begin{aligned}
(xs ::: ys) ::: zs &= xs ::: (ys ::: zs) \\
xs ::: List() &= xs = List() ::: xs
\end{aligned}
$$

Q: How can we prove statements like the one above?

A: By structural induction over lists.

Reminder: Natural Induction

Recall the proof principle of natural induction:

To show a property $P(n)$ for all numbers $n \geq b$:

1. Show that $P(b)$ holds (base case).
2. For arbitrary $n \geq b$ show:

   if $P(n)$ holds, then $P(n+1)$ holds as well

   (induction step).

Example: Given

```
def factorial(n: int): int =
    if (n == 0) 1
    else n * factorial(n-1)
```

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

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

Case *4* is established by simple calculation of $factorial(4) = 24$ and $2^4 = 16$.

Case *n+1* We have for $n \geq 4$:

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

Note that in our proof we can freely apply reduction steps such as in (*) anywhere in a term.

This works because purely functional programs do not have side effects; so a term is equivalent to the term it reduces to.

The principle is called *referential transparency*.

Structural Induction

The principle of structural induction is analogous to natural induction:

In the case of lists, it is as follows:

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

1. Show that $P(\mathit{List}(\,))$ holds (base case).
2. For arbitrary lists xs and elements x show:

   if $P(xs)$ holds, then $P(x :: xs)$ holds as well

   (induction step).

Example

We show $(xs ::: ys) ::: zs \;=\; xs ::: (ys ::: zs)$ by structural induction on xs.

Case $List\,()$ For the left-hand side, we have:

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

For the right-hand side, we have:

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

So the case is established.

Case $x :: xs$

For the left-hand side, we have:

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

For the right-hand side, we have:

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

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

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

Example (2)

As a more difficult example, consider function

$$
\begin{array}{l}
\textbf{\textit{abstract class}}\ \mathit{List}\,[\mathrm{a}]\ \{\ \ldots \\
\quad \textbf{\textit{def}}\ \mathit{reverse}: \mathit{List}\,[\mathrm{a}] = \mathit{match}\ \{ \\
\quad\quad \textbf{\textit{case}}\ \mathit{List}\,(\,) \Rightarrow \mathit{List}\,(\,) \\
\quad\quad \textbf{\textit{case}}\ x :: \mathit{xs} \Rightarrow \mathit{xs.reverse} ::: \mathit{List}\,(x) \\
\quad \} \\
\}
\end{array}
$$

We would like to prove the proposition that

$$\mathit{xs.reverse.reverse} \;=\; \mathit{xs} \quad .$$

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

$$
\begin{array}{ll}
 & \mathit{List}\,(\,).\mathit{reverse.reverse} \\
= & \text{(by first clause of } \mathit{reverse}\text{)} \\
 & \mathit{List}\,(\,).\mathit{reverse} \\
= & \text{(by first clause of } \mathit{reverse}\text{)} \\
 & \mathit{List}\,(\,)
\end{array}
$$

For the induction step, we try:

$$
\begin{array}{ll}
 & (x :: xs).reverse.reverse \\
= & (\textit{by second clause of } reverse) \\
 & (xs.reverse ::: List(x)).reverse
\end{array}
$$

There's nothing more we can do to this expression, so we turn to the right side:

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

The two sides have simplified to different expressions.

So we still have to show that

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

Trying to prove this directly by induction does not work.

Instead we have to *generalize* the equation to:

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

This equation can be proved by a second induction argument over ys. (See blackboard).

Exercise: Is it the case that $(xs\ drop\ m)\ at\ n \ = \ xs\ at\ (m + n)$ for all natural numbers m, n and all lists xs?

Structural Induction on Trees

Structural induction is not restricted to lists; it works for arbitrary trees.

The general induction principle is as follows.

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

- Show $P(l)$ for all leaf trees l.
- For every interior node t with subtrees s_1, ..., s_n, show that $P(s_1) \wedge \ldots \wedge P(s_n) \Rightarrow P(t)$.

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

```
abstract class IntSet {
  abstract def incl(x: int): IntSet
  abstract 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: Set, right: Set) 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*** added, so that we can use factory methods instead of ***new***).

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

Laws of IntSet

One way to state and prove the correctness of an implementation is to prove laws that hold for it.

In the case of *IntSet*, three such laws would be:

For all sets s, elements x, y:

$$
\begin{array}{llll}
\textit{Empty contains } x & = & \textbf{\textit{false}} & \\
(s \textit{ incl } x) \textit{ contains } x & = & \textbf{\textit{true}} & \\
(s \textit{ incl } x) \textit{ contains } y & = & s \textit{ contains } y & \textbf{\textit{if}}\ x \neq y
\end{array}
$$

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

How can we establish that these laws hold?

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

Proof: By the definition of *contains* in *Empty*.

Proposition 2: $(xs\ incl\ x)\ contains\ x = \boldsymbol{true}$

Proof:

Case *Empty*

$$
\begin{array}{ll}
& (Empty\ incl\ x)\ contains\ x \\
= & (\text{by definition of } incl \text{ in } Empty) \\
& NonEmpty(x, Empty, Empty)\ contains\ x \\
= & (\text{by definition of } contains \text{ in } NonEmpty) \\
& \boldsymbol{true}
\end{array}
$$

Case *NonEmpty(x, l, r)*

$$
\begin{array}{ll}
& (NonEmpty(x, l, r)\ incl\ x)\ contains\ x \\
= & (\text{by definition of } incl \text{ in } NonEmpty) \\
& NonEmpty(x, l, r)\ contains\ x \\
= & (\text{by definition of } contains \text{ in } Empty) \\
& \boldsymbol{true}
\end{array}
$$

Case *NonEmpty* (y, l, r) where $y < x$

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

Case *NonEmpty* (y, l, r) where $y > x$ is analogous.

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

Proof: See blackboard.

Exercise

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

```
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 subsumed with the following law:

Proposition 4: (*xs union ys*) *contains x* = *xs contains x* || *ys contains x*.

Is that true ? What hypothesis is missing ? Show a counterexample.

Show Proposition 4 using structural induction on *xs*.