EPFL

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.

```scala
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) ...))
```

They are defined as follows

```scala
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:

```scala
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:

```scala
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
```

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

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

```
  Nil

=   reverse(Nil)
```

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`

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)
```

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?)
```

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.

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 ?

We thus arrive at the following implementation of reverse.

```scala
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:

```scala
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)
```