Generation, Iteration, Traversal of Combinatorial Objects

Some Musings on Permutations

Reduce and Conquer

There is a well known and intuitive reduce-and-conquer algorithm for generating all permutations:

\[
\text{perms}(l) = \begin{cases} 
\text{List}(\text{Nil}) & \text{if } l = \text{Nil}, \\
\text{Ins}(x, \text{perms}(l')) & \text{if } l = x :: l'
\end{cases}
\]

The list of all permutations of the empty list is a list with one element, the empty list. All permutations of a list that was created by adding an element \(x\) to a list \(l'\) are computed by computing all permutation of \(l'\) and then computing all possible insertions of \(x\) into the permutations of \(l'\).

For computing all possible insertions of an element \(x\) into each member of a list of lists \(L\) there is also a simple reduce-and-conquer algorithm:

\[
\text{Ins}(x, L) = \begin{cases} 
\text{Nil} & \text{if } L = \text{Nil}, \\
\text{ins}(x, l) :: \text{Ins}(x, L') & \text{if } L = l :: L'
\end{cases}
\]

In order to compute all insertions of an element \(x\) into all lists \(l\) of list a list of lists \(L\) we loop over the elements \(l\) of \(L\), compute all insertions of \(x\) into \(l\) and join all results into one list.

Function \(\text{Ins}\) operates on lists of lists and needs a helper function \(\text{ins}\) which operates on lists. Function \(\text{ins}\) computes all insertions of an element into a list. For \(\text{ins}\) too, we have a beautiful reduce-and-conquer algorithm:

\[
\text{ins}(x, l) = \begin{cases} 
\text{List}(\text{List}(x)) & \text{if } l = \text{Nil}, \\
(a :: l') :: \text{List}(a :: l' | l' \in \text{ins}(x, l')) & \text{if } l = a :: l'
\end{cases}
\]

All insertions of an element \(x\) into a list \(l\) with first element \(a\) and a rest \(l'\) are computed by computing all insertions of \(x\) into \(l'\) and prepending \(a\) to each of them. Finally add the list \(a :: l\) to all of them.

Origami: Mapping and Folding

The definitions above may be transferred to code nearly directly:

```scala
def perms(l: List[Symbol]) : List[List[Symbol]] = l match {
  case Nil => List(Nil)
  case x::ll => Ins(x , perms(ll))
}

def Ins(x: Symbol, L: List[List[Symbol]]) : List[List[Symbol]] = L match {
  case Nil => List(Nil)
  case l::ll => Ins(x , perms(ll)) :: Ins(x , Ins(x, l)) :: Ins(x , Ins(x, l))
}
```
case Nil \Rightarrow Nil
\text{case } l::rL \Rightarrow \text{ins}(x, l) :::: \text{Ins}(x, rL)
\}

\text{def } \text{ins}(x: \text{Symbol}, l: \text{List}[\text{Symbol}]) : \text{List}[\text{List}[\text{Symbol}]] = l \text{ match } \\
\text{case Nil } \Rightarrow \text{List(}\text{List}(x)) \\
\text{case } a::rl \Rightarrow \text{x::l) :: (ins}(x, rl) \text{ map}(a :: _) } 
\}

A short look at them reveals that one of these functions \text{Ins} does nothing more than applying a further function \text{ins} to all elements of a list and joining the results of \text{ins} to single list. This leads us to the following redefinition of \text{Ins}:

\text{def } \text{Ins}(x: \text{Symbol}, L: \text{List}[\text{List}[\text{Symbol}]])) : \text{List}[\text{List}[\text{Symbol}]] = \\
L.\text{map( } \text{ins}(x, _) ) \text{ reduce(_:::_).}

This function has a body that is shorter than its header and it is easy to understand: it is just a loop on all elements of \text{L}. So we remove \text{Ins} completely. This leads us to:

\text{def } \text{perms}(l: \text{List}[\text{Symbol}]) : \text{List}[\text{List}[\text{Symbol}]] = l \text{ match } \\
\text{case Nil } \Rightarrow \text{List(Nil)} \\
\text{case } x::ll \Rightarrow \text{(perms}(ll) \text{ map( ins}(x, _) )) \text{ reduce(_:::_);} 
\}

For mapping and subsequently reducing with \text{::} there is a special operator: \text{flatMap}. With \text{flatMap} we get our final version of the reduce-and-conquer permutation generation:

\text{def } \text{perms}(l: \text{List}[\text{Symbol}]) : \text{List}[\text{List}[\text{Symbol}]] = l \text{ match } \\
\text{case Nil } \Rightarrow \text{List(Nil)} \\
\text{case } x::ll \Rightarrow \text{(perms}(ll) \text{ flatMap( ins}(x, _) ))
\}

\textbf{Traversing and Iterating Permutations}

Permutations are typical combinatorial objects that may be used in an exhaustive search or any other computation that performs an operation on each of them. There are two basic patterns for the examination of all elements of a collection: traversal and iteration. Traversal is performing an operation on all elements of a collection under the control of the collection. Iteration is performing an operation on all elements of a collection under the control of its user.

Traversable collections have a method \text{foreach(action)} which takes an action parameter that is applied to all elements of the collection. The permutations may be transformed into a traversable collection by defining the \text{foreach} method. In order to retain the functional trait we define a case class that extends \text{Traversable}:

\text{case class } \text{Perms}(l: \text{List}[\text{Symbol}]) \text{ extends } \text{Traversable}[\text{List}[\text{Symbol}]] \\
\text{def } \text{foreach}[U](f: \text{List}[\text{Symbol}] \Rightarrow U): \text{Unit} = \\
\quad \text{// apply } f \text{ to each element discarding its result of type } U
\}

2
A simple and obvious implementation is to generate all permutations as a list and use it. Lists are traversable.

```scala
package object traverse_and_iterate {

case class Perms(l: List[Symbol]) extends Traversable[List[Symbol]] {
  val lst: List[List[Symbol]] = perms(l: List[Symbol])
  def foreach[U](f: List[Symbol] => U): Unit = lst.foreach(f)
}

// unchanged helpers:
private def perms(l: List[Symbol]) : List[List[Symbol]] = ...;
private def ins(x: Symbol, l: List[Symbol]) : List[List[Symbol]] = ..
}
```

An example of usage is:

```scala
Perms(List('a, 'b, 'c)).foreach { x => println(x) }
```

The insertions can be made traversable using the same technique:

```scala
case class Ins(x: Symbol, l: List[Symbol]) extends Traversable[List[Symbol]] {
  val lst: List[List[Symbol]] = ins(x, l)
  def foreach[U](f: List[Symbol] => U): Unit = lst.foreach(f)
}
```

All this is completely superfluous: the lists that are produced by the original functions are traversable, so there is no need to define a new class:

```scala
perms(List('a, 'b, 'c)).foreach { x => println(x) }
// same as Perms(List('a, 'b, 'c)).foreach { x => println(x) }
```

A list is traversable and thus may be used when a traversable object is needed. But a traversable object is not necessarily a list, and thus may not replace a list. So let’s consider the situation when we have a traversable object instead of a list. How do we define permutation generation if we just have the insertions as traversable object. Remember the original version of the function `perms`:

```scala
def perms(l: List[Symbol]) : List[List[Symbol]] = l match {
  case Nil => List(Nil)
  case x::ll => (perms(ll) map( ins(x, _) )) . reduce(_:::_)
}
```

Instead of `ins` we just have `Ins`. A list may be used by a function, a traversable object uses a function. So if we have `Ins(...) (a traversable object) instead of `ins(...) (a list), we have to somehow turn the control structure inside out:

```scala
import scala.collection.mutable.ListBuffer
def perms(l: List[Symbol]) : List[List[Symbol]] = l match {
  case Nil => List(Nil)
  case x::ll => {
    var res : ListBuffer[List[Symbol]] = new ListBuffer()
    perms(ll).foreach { p => Ins(x, p).foreach { l => res += l } }
    res.toList
  }
}
```
We traverse the insertions and collect them into a list. However if we do not want to generate a list of all permutations but just provide a way to traverse them, we do not have to collect them, and this version is appropriate:

```scala
case class Perms(l: List[Symbol]) extends Traversable[List[Symbol]] {
  def foreach[U](f: List[Symbol] => U): Unit = l match {
    case Nil => f(Nil)
    case x::ll =>
      Perms(ll).foreach {
        p => Ins(x, p).foreach { l => f(l) }
      }
  }
}
```

Iterating though all permutations differs from traversing them in that the user controls the process of going through all elements, instead of the collection itself. This means that control switches between user code and collection code. Each time control leaves the collection the current state of traversing the elements has to be saved and when it returns this state has to be resumed.

Saving and resuming the current state of a traversal is easy in the case of a list. The state is just the current position in the list. It is more complex when we go through a tree because the actual position does not determine the next position: next we may have to go left or right down or even up.

All permutations are a list of lists and lists are iterable, so creating an iterator for all permutation is trivial as creating a traversable object:

```scala
case class Perms(l: List[Symbol]) extends Iterable[List[Symbol]] {
  def iterator = perms(l).iterator

  private def perms(l: List[Symbol]) : List[List[Symbol]] = l match {
    case Nil => List(Nil)
    case x::ll => (perms(ll) map( ins(_, _) )) . reduce(_:::_)
  }

  private def ins(x: Symbol, l: List[Symbol]) : List[List[Symbol]] = l match {
    case Nil => List(List(x))
    case a::rl => (x::l) :: (ins(x, rl) map(a :: _))
  }
}
```

A more appropriate and elegant solution uses recursive functions that produce iterators:

```scala
def permsIter(l: List[Symbol]) : Iterator[List[Symbol]] = l match {
  case Nil => Iterator(Nil)
  case x::ll => (permsIter(ll) map( insIter(_, _) )) . reduce(_++_)
}

def insIter(x: Symbol, l: List[Symbol]) : Iterator[List[Symbol]] = l match {
  case Nil => Iterator(List(x))
  case a::rl => Iterator(x::l) ++ (insIter(x, rl) map(a :: _))
}

for ( l <- permsIter(List('a, 'b, 'c, 'd, 'e))) { println(l) }
```
Generating Permutations on the Fly

A list of \( n \) elements has \( n! \) permutations. This might be too much for the heap. If we have to inspect them all, e.g. as part of an exhaustive search algorithm, it would be better to generate one after the other. Our simple iterable and traversable objects could give the impression of producing one element after the other, but all them first produce all permutations (or at least all permutations of a list that is one element shorter) and then go through them.

In order to produce all permutations on the fly, we have the following alternatives:

- Use the reduce-and-conquer algorithm and find a way to halt and resume the recursive algorithm
- Invent a completely different algorithm that can go form one permutation to the next.

Let’s consider the first alternative first. We have to stop and resume a recursive algorithm. Recursive algorithms use a stack to store the actual state. So we have to find a way to store and later re-install the current stack. There are two classical techniques to that allow us to deal with the stack:

- threads, and
- continuations

Threads automatically store and resume the stack, when entering and leaving blocking operations. Continuations are a technique that allows us to deal explicitly with “the rest of the computation” and this is rest ist represented by the stack.

The easiest way to stop and resume any computation, recursive or not, is using a thread. Whenever a permutation is produced, we stop the thread and hand the permutation to the user. Now we have a producer–consumer scenario. One thread produces the permutations, the other consumes it. The producer may be dressed as iterator:

```java
import java.util.concurrent.{BlockingQueue, ArrayBlockingQueue}

case class IterablePerms(l: List[Symbol]) extends Iterable[List[Symbol]] {
  val q: BlockingQueue[List[Symbol]] = new ArrayBlockingQueue(l)
  var countPerms = 0
  val fact: Int => Int = {(n:Int) => if (n==0) 1 else n*fact(n-1) }
  val limit: Int = fact(l.length)

  def iterator: Iterator[List[Symbol]] = new Iterator[List[Symbol]] {
    def hasNext: Boolean = countPerms < limit
    def next: List[Symbol] = {
      countPerms = countPerms+1
      q.take()
    }
  }

  val t = new Thread(
    new Runnable {
      override def run() {
        TraversablePerms(l: List[Symbol]).foreach { x => q.put(x) }
      }
    }
  )
  t.start()
}
```

```java
case class TraversablePerms(l: List[Symbol]) extends Traversable[List[Symbol]] {
  def foreach[U](f: List[Symbol] => U): Unit = l match {
    case Nil => f(Nil)
  }
}
```
Threadsm are a rather heavy tool. There are more lightweight mechanisms to realize this kind of coroutines. Python has introduced a special language feature called generators. A lightweight (but mentally rather heavy) language independent mechanism to realize arbitrary control structures are continuations. Scala does not have generators, instead it supports continuations and a further feature called streams. Streams – and eventually continuations – are considered to be an alternative to generators. Usage of continuations isn’t supposed to be a business of mere mortals. It should be avoided.¹ So let’s have a look at streams.

**Streaming Permutations**

Streams are lazy lists that produce elements only on demand. As an example of how to use them, the Scala API shows how to stream fibonacci numbers. We will start with a simple example, a stream of three natural numbers, which behaves exactly as a list of three numbers:

```scala
import Stream._ // cons and Empty are static members of Stream
val str = cons(1, cons(2, cons(3, Empty)))
for (i <- str) {
  println(i)
}
```

For the convenience of the user there are implicit conversions to ConsWrappes with an #:: operator that make streams look like lists:

```scala
val str = 1 #:: 2 #:: 3 #:: Stream.Empty
for (i <- str) {
  println(i)
}
```

A stream consists of cells that store functions that produce the head and the rest of the stream. The code

```scala
println("construction")
val str = {println(a); 1} #:: {println(b); 2} #:: {println(c); 3} #::
  Stream.Empty
println("usage")
for (i <- str) {
  println(i)
}
println("second usage")
for (i <- str) {
  println(i)
}
```

¹ Unless you feel intellectually challenged by them and want demonstrate your superiority by writing yet another blog on continuations.
will produce the output:

```
construction
  'a
usage
  1
  'b
  2
  'c
  3
second usage
  1
  2
  3
```

showing that the cells are evaluated on demand and that computed values will remain in evaluated form in the stream.

This may be used to create infinite streams:

```
val nats = {
  def generator(n: Int) : Stream[Int] = n #: generator(n+1)
  generator(1)
}
for (n <- nats) {
  println(n)
}
```

or to create streams that produce values only on demand, even if there are not infinitely many of them.

Let’s apply these mechanisms to the permutations. This is surprisingly easy: just a minor transformation of the original reduce-and-conquer function:

```
import Stream.Empty

def perms(l: List[Symbol]) : List[List[Symbol]] = l match {
  case Nil => List(Nil)
  case x::ll => (perms(ll) flatMap( ins(x, _) ))
} // is transformed to:

def permS(l: List[Symbol]) : Stream[List[Symbol]] = l match {
  case Nil => Nil #:: Empty
  case x::ll => (permS(ll) flatMap( inS(x, _) ))
}
def ins(x: Symbol, l: List[Symbol]) : List[List[Symbol]] = l match {
  case Nil => List(List(x))
  case a::rl => (x::l) :: (ins(x, rl) map(a :: _))
} // is transformed to:

def inS(x: Symbol, l: List[Symbol]) : Stream[List[Symbol]] = l match {
  case Nil => List(x) #:: Empty
  case a::rl => (x::l) #:: (inS(x, rl) map(a :: _))
}
```
**Generating Tuples**

Using implementations of the original reduce-and-conquer algorithm, we have to generate all permutations before we can use the first one. Using any (!) of the other versions including the streaming and threading versions (and the python–generator version too) improve this by just one level: all permutation of a list that is one element shorter have to generated before we can produce the first permutation. This reduces storage demand from \( n! \) to \((n - 1)!\) – Not that impressive.

All versions that we considered up to now were based on the reduce-and-conquer algorithm, with its essential feature that the list of permutations of length \( n \) are computed recursively using permutations of \( n - 1 \) elements. So it is obvious that we may throw as we like threads, generators, iterators, streams etc. into the scene, this will not improve the situation. We need a really different algorithm. A really sequential algorithm that starts with a first permutation form which the next may be produced and so on. In short, we need an algorithm that enumerates permutations.

In a draft section\(^2\) of his *The Art of Computer Programming*, eternally *in statu nascendi*, D. Knuth presents an algorithm that enumerates all permutations. We will not follow this algorithm here, but look at a simpler but more expensive version. First we observer that permutations are related to tuples.

A tuple of length \( k \) (also called \( k \)-mer) has \( k \) positions which are occupied with elements form a set \( \Sigma \). There are \( k^n \) tuples of length \( k \) over a set of size \( n = |\Sigma| \). For the permutations of a string \( s \) let’s assume that all elements of \( s \) are different. (You may think of them as indexed by natural numbers and \( s \) replaced by the sequence of indices.)

The number of permutations is \( n! \) if \( s \) has length \( n \).

Now think of the tuples that we can build using the (pairwise distinct) elements of \( s \) (or their indices). There are \( n^k \) different tuples of length \( k \) with elements taken from a set of size \( n \). If \( n = k \) there are \( n^n \) tuples. Some of these tuples are permutations. \( n^n \) is considerably larger than \( n! \) because a lot of the tuples are not permutations: all tuples that contain repeated elements are not permutations. If we generate all tuples and sort out all those that contain repetitions, we are done.

Generating all tuples can be done with a kind of counter. Assume we have to generate all tuples of a set of elements \( s_0, s_1, \ldots, s_{n-1} \). We assume that \( s_0 < s_1 < \cdots s_{n-1} \) and start with \( s_0, s_0, \ldots, s_0 \): the "counter" is set to 00\ldots 0.

Then we increase the "counter" step by step.

\[
\begin{align*}
000 \cdots 0 \\
000 \cdots 1 \\
\ldots \\
000 \cdots n - 1 \\
00 \cdots 10 \\
\ldots \\
nnn \cdots n
\end{align*}
\]

Thus we may generate all \( n^n \) tuples. There is a very simple procedure to generate these tuples, nested loops:

```scala
val lst: List[Symbol] = List('a, 'b, 'c)

for(s1 <- lst;
s2 <- lst;
s3 <- lst) {
  println(List(s1, s2, s3))
}
```

Of course this will not work if the size of \( s \) is known only at runtime. If it is not, we will have to code the ellipsis (the "dots") of

```scala
for(s1 <- lst;
   ....
   sn <- lst) {
  println(List(s1, .... sn))
}
```

---

\(^2\) A Draft of section 7.2.1.2: Generating all Permutations, http://www-cs-faculty.stanford.edu/~uno/taocp.html
This can be done using recursion:

```scala
def allTuples[T](k: Int, s: List[T]): List[List[T]] = {
    def tuples(k: Int) : List[List[T]] = {
        if (k == 0) List(Nil)
        else for (t <- tuples(k-1); b <- s) yield t :+ b
    }
    tuples(k)
}
def allTuples[T](s: List[T]): List[List[T]] = allTuples[T](s.length, s)

val lst: List[Symbol] = List('a, 'b, 'c)
for( t <- allTuples(lst)) {
    println(t)
}
```

This algorithm has the same deficiency as our permutation algorithms: tuples are generated recursively using all sorter tuples. So let's again consult D. Knuth for this problem. We find in another draft section of his *The Art of Computer Programming,* 3 Here we find algorithm M:

M1. [Initialize.] Set \( a_j \leftarrow 0 \) for \( 0 \leq j \leq n \), and set \( m_0 \leftarrow 2 \).

M2. [Visit.] Visit the \( n \)-tuple \((a_1, \ldots, a_n)\). (The program that wants to examine all \( n \)-tuples now does its thing.)

M3. [Prepare to add one.] Set \( j \leftarrow n \).

M4. [Carry if necessary.] If \( a_j = m_j - 1 \), set \( a_j \leftarrow 0 \), \( j \leftarrow j - 1 \), and repeat this step.

M5. [Increase, unless done.] If \( j = 0 \), terminate the algorithm. Otherwise set \( a_j \leftarrow a_j + 1 \) and go back to step M2.

This may be coded as:

```scala
case class AllTuples[T](k: Int, l: List[T]) extends Traversable[List[T]] {
    def foreach[U](f: List[T] => U) : Unit = {
        var a : Buffer[Int] = Buffer.fill(k)(0)
        var j = 0
        var stop = false
        while (!stop) {
            f( a.map(l(_)).toList )
            j = k-1
            while (j >= 0 && a(j) == l.length-1) {
                a(j) = 0; j = j-1
            }
            if (j>=0) { a(j) = a(j)+1 } else { stop = true }
        }
    }
}
```

3 A Draft of section 7.2.1.1: Generating all Tuples, http://www-cs-faculty.stanford.edu/~uno/taocp.html
Generating Permutations by Filtering Tuples

Now let’s turn again to permutations. The set of all permutations is a subset of all tuples: it consists of those tuples that do not contain repetitions, provided the list does not contain repetitions, which we may assume without restricting our solution, because we can always identify elements with their position in the list. This leads immediately to traversable permutations that are generated on the fly:

```scala
import scala.collection.mutable.Buffer

case class AllPerms[T](l: List[T]) extends Traversable[List[T]] {
  val k = l.length
  def foreach[U](f: List[T] => U) : Unit = {
    var a : Buffer[Int] = Buffer.fill(k)(0)
    var j = 0
    var stop = false
    while (!stop) {
      if (a.distinct.length == k) { // <= no repetitions in a ?
        f( a.map(l(_)).toList )
      }
      j = k-1
      while (j >= 0 && a(j) == l.length-1) {
        a(j) = 0
        j = j-1
      }
      if (j>=0) {
        a(j) = a(j)+1
      } else {
        stop = true
      }
    }
  }
}
```

This algorithm is our first one that will not fill the memory before producing the first permutation. It is simple and rather fast.

We observe that we will have to wait a long time before the first permutation is produced. This is not a surprise because we fill the “counter” `a` with 0-s and then increase it until all digits are different. It is easy to speed this up, just start with the first permutation:

```scala
case class AllPerms[T](l: List[T]) extends Traversable[List[T]] {
  val k = l.length
  def foreach[U](f: List[T] => U) : Unit = {
    var a : Buffer[Int] = Buffer.tabulate(k)((i:Int) => i)
    // ... unchanced ...
  }
}
```

At this point we stop our musings on permutations. For those who want to dwell on the topic, there is a wealth of fascinating material that may keep them busy for the rest of their lives. For those who just need an efficient permutation enumerator there is Google.