8 Shortest-strings Problem: Partial-Solution Dump

Drew McDermott drew.mcdermott@yale.edu

Lecture notes for 2016-09-19, revised 20016-09-25

Here is the state of play in solving the little puzzle about finding the shortest strings in a list of lists of strings, i.e., the list of strings that are the shortest in one of the component lists of the input, y1.[1]

I didn't in class, but it seems cleaner now to treat y as the parameter to a function:

```
/** Find the strings that are the shortest in some list in `y`
  */
def shortestListsRecur(y: List[List[String]]): List[String] = {
  def findShortest(shortestSoFar: List[String], // Nonempty
                   l: List[String]): List[String] = l match {
    case Nil => shortestSoFar
    case s :: ss => {
      val lengthToBeat = shortestSoFar.head.length
      if (s.length < lengthToBeat) ...
      else if (s.length == lengthToBeat) ...
      // The champ wins again
      else findShortest(shortestSoFar, ss)
    }
  }
// Body of shortestLists starts here
y flatMap {
    case Nil => Nil
    case x :: xs => findShortest(List(x), xs)
  }
}
```

I've also switched the order of the two parameters of findShortest, for reasons that will become clear.

Exercise: Finish shortestListsRecur.

Many recursions through a list (call it L) exhibit one of these two patterns (see figure 1):

1. *Left recursion:* A "value so far" is passed as an argument as successive elements of the list L are examined and used in a computation of the element to be passed to the next stage.

[1] The resulting list may well have duplicate elements, if the same string is shortest in more than one list.

2. *Right recursion:* The value returned from applying the function to the tail of the list is combined with the head of the list in a computation of the value to be returned from this stage.

In each case, the function must be given an initial value to start the computation with. For left recursion, this is the first value of the "result so far" passed as an argument to the function. For right recursion, it's also the first "result so far," but now injected into the chain of computations as the value returned when the function encounters the `Nil` at the end of every list.[2]

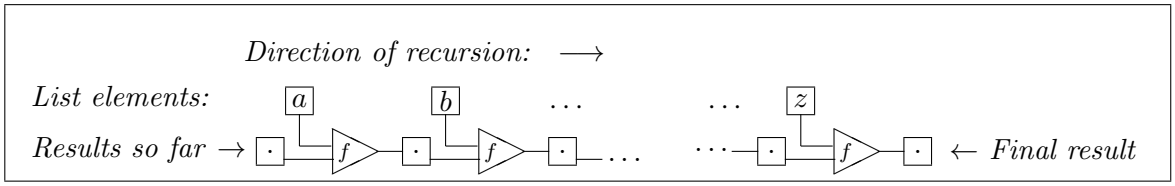
Many functions can be defined using either left or right recursion. Here is a right-recursive version of `findShortest` and how it would be called:

```
def shortestListsRRecur(y: List[List[String]]): List[String] = {
  Same function as before, with right recursion
  def findShortest(l: List[String]): List[String] = l match {
    case Nil => Nil
    case s :: ss => findShortest(ss) match {
      case Nil => List(s)
      case h :: hs => {
        val lengthToBeat = h.length
        if (s.length < lengthToBeat) ...
        else if (s.length == lengthToBeat) ...
        else h :: hs
      }
    }
  }
```

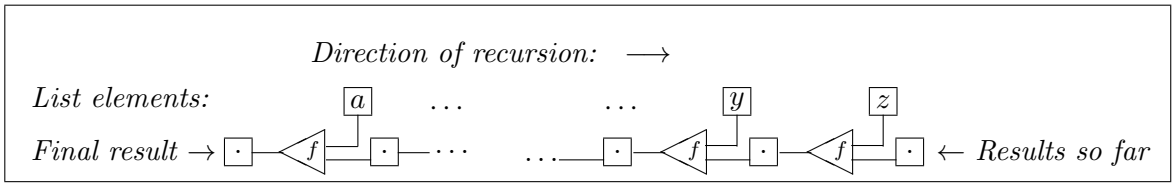
The pieces left out here are essentially the same as those left out of the left-recursive version. Rather than having to repeat this code, let's package them up as a subroutine bound in some appropriate place:

```
def chooseNewChamps(champsSoFar:  List[String], newCand:  String) = {
  val candLen = newCand.length
  val shortestLen = champsSoFar.head.length
  if (candLen < shortestLen) ...
```

[2]Again, these patterns work for other collections, with a couple of obvious caveats. Each depends on getting through the whole collection, so neither will work for an infinite stream. (See file `StreamDemo.scala`, lecture 9.) Each depends on being able to break a nonempty collection into a "head" and a "tail." If the collection is unordered, we can select some arbitrary element as the head and the rest of the collection as the tail, but some functions will return different values depending on which element is selected at each stage.



(a) Left Recursion



(b) Right recursion

Figure 1: *Left and Right Recursion:* The list elements are the same, left-to-right, and direction of recursion is the same. The function f is the computation done at each stage to combine the next list element with the *result so far* to yield the *result so far* at the next stage. (The computation f is the same at each stage, but of course f in pattern (a) is not neacessarily the same as f in pattern (b).) In part (a), the computation must be "primed" with an initial value for the *result so far.* In part (b), it must be primed with a *result so far* for the base case (when the list = `Nil`).

```
    else if (candLen == shortestLen) ...
    else
    champsSoFar
  }
```

Now `shortestListsRRecur` can be defined more economically as

```
  def shortestListsRRecur(y: List[List[String]]): List[String] = {
    Same function as before, with right recursion
    def findShortest(l: List[String]): List[String] = l match {
      case Nil => Nil
      case s :: ss => findShortest(ss) match {
        case Nil => List(s)
        case shortestFromRight => chooseNewChamps(shortestFromRight, s)
      }
    }
//
y flatMap findShortest
}
```

Instead of passing an extra value "down," we're passing a value "up," i.e., returning it. But we're taking a different approach. Instead of catching the special case of an empty list at the top, before calling `findShortest`, we use `Nil` as the initial result-so-far. If the entire list is empty, this will be the final value. Otherwise, `foldRight` will see `Nil` only when looking at the next-to-last tail of the list, when `ss` = `Nil`. In that case, the head `s` is the first "champ" to beat. As successive elements are examined from right to left, the current front-runner must beat all of them, or be deposed and replaced.

The only problem with this idea binding it to the `val shortestFromRight`, named to highlight the symmetry with the parameter `shortestSoFar` in the left-recursive version, which we could have called `shortestFromLeft`). The only weird part is that the base case is a list of length one rather than `Nil`; the reason for this oddness is that `findShortest` has to be "primed" with a first guess as to the shortest string so far. In the left-recursive version the first guess will be `List(y.head)`. Here it's a list of the *last* element of `y` (the only argument `findShortest` ever actually gets). Either way, `findShortest` will not work with an empty list, so we'll have to treat it specially.
Exercise: Finish this version of `shortestLists`.

Any repeated pattern of behavior should be abstractable. That is, we should be able to write a function that captures the repeated skeleton of

the behavior, with parameters that fill in the parts that vary from one occurrence of the behavior pattern to another. The results in this case are called `foldLeft` and `foldRight`. Each has three parameters: l, the list[3] being traversed, z, an initial value, and f, a function of two arguments. This last parameter takes the "result so far" r and the next element x of l and produces the "result after seeing x." For mnemonic purposes, for `foldLeft` f takes the r and x in that order, while `foldRight` takes them in the order `(`x`, `r`)`. The initial value z and the result so far r need not be the same type as the elements of l, and for `findShortest` they aren't. The elements of the list are of type `String` and the intermediate (and final) results are of type `List[String]`.

Here is the way we could define `findShortest` using `foldLeft`:[4]

```
def findShortest(l: List[String]) =
  l.tail.foldLeft(l.head)((shortestSoFar: List[String], s: String) =>
    val lengthToBeat = shortestSoFar.head.length
    if (s.length < lengthToBeat) ...
    else if (s.length == lengthToBeat) ...
    // The champ wins again
    else shortestSoFar
  })
```

The parameter l in the schema above is "passed" as the 0'th argument by putting it to the left of the dot before `foldLeft`. The function `foldLeft` has two "true" parameters, z and f, but they are in separate parameter lists.[5] The parameter z is the sole occupant of list 1, and f is the sole occupant of list 2. Instead of writing `l.tail.foldLeft(z, f)`, you have to remember to write `l.tail.foldLeft(z)(f)`. Actually, Scala so often "curries" its built-in functions when a functional parameter is involved that one comes to expect it.

This may not look like much of an improvement, and perhaps it isn't. But now `findShortest` isn't recursive, so we may not need it at all. It's called only once. One has to use one's judgment here as always whether to

[3] Actually, any collection. The output collection is of the same type as the input, when possible. See Lecture Note 7 for the details.

[4] Even though we also defined a version of `findShortest` using right recursion, it was "deviant," in the sense that it terminated when it got down to a list of length one. The standard version of `foldRight` abstracts the pattern of behavior in which the recursion's base case is the empty list. This deviance is enough to make `foldRight` inapplicable without some trickery. See the [[exercise below.]]

[5] For reasons best left unexplained at this point, but if you insist, reread section 3.3.2 of FPS.

define a subroutine or skip it. It depends on issues like whether the function passed as the last argument is big and messy or compact and clear. In the current case, it's big enough and messy enough that having a name for the function, and commenting it judiciously, makes sense. And if it were up to me, I would move the test for the special case of an empty list inside the function definition, since it's messy already:

```
def findShortest(l:List[String]):List[String] = {
  if (l == Nil) List()
  else l.tail.foldLeft(l.head)((...)  =>
    ...)
}
```

In case `foldLeft` and `foldRight` are not obscure enough for you, you can abbreviate them using `/:` and `:\`, respectively:

$$\begin{aligned} (x\ \texttt{/:}\ l)(f) &= l.\texttt{foldLeft}(x)(f) \\ (l\ \texttt{:\textbackslash}\ x)(f) &= l.\texttt{foldRight}(x)(f) \end{aligned}$$

The astute reader will notice that the arguments occur in a different order in these two constructions. The mnemonic is the COLlection and COLon are on the same side. Scala has very simple rules for determining the precedence and associativity (right- or left-) of an operator:

1. The precedence of an operator depends on its first character. Any operator beginning with `'*'` has the same precedence as "`*`" ("times"). The full ordering is

   $$\textit{letters} \prec \texttt{|} \prec \texttt{\^{}} \prec \texttt{<,>} \prec \texttt{=,!} \prec \texttt{:} \prec \texttt{+,-} \prec \texttt{*,/,\%} \prec \textit{others}$$

   where *others* means all other special characters.[6]

2. Operators ending in colon are right-associative. Further, when the dots are left out, i.e., when you write "`a #%%: b`", it means "`b.#%%:(a)`"; whereas it would be a call to a member of `a` if there's no colon, so "`a #%% b`" is parsed as "`a.#%%(b)`".

3. *Assignment operators* have different rules. These are '`=`' plus any multicharacter operator ending in '`=`', unless it either starts with an '`=`'

[6] From the *Scala Language Specification, v. 2.9*, Martin Odersky, 2014, sec. 6.12.3.

(e.g., '`=#%&=`') or can be interpreted as a comparison (e.g., '`>=`'). Assignment operators have lower precedence than all other operators. If `###` is an operator, then e_1`###=`e_2 is short for e_1`=`e_1`###`e_2 except that presumably the compiler only evaluates the subexpressions of e_1 once. and suppose `++` is defined by

```
def ++():  Int { j = i; i = j+1; j }
```

(where `i` is bound in the scope of `++`). If `a` is an array, then `a(++()) *= b(i)` is compiled as though it were

```
{ val j99 = ++(); a(j99) = a(j99) * b(i)}
```

so that `i` is incremented just once.

Exercise: Although `foldRight` can't be applied directly, a little trickery will make it applicable. One approach is to divide the list into the last element and a new list of all the elements before the last one. This is ugly; we have to traverse the list twice and build a list we don't really need. Perhaps we overlooked a *result so far* we could return for `Nil` that would allow us to write an f function for the right recursion.

[[One might make the following suggestion: Why not use the first element of the list as the first *result-so-far* (call it Z) even when "folding right"? [[show code]] Because if we know the following facts about f and Z (let `A` be the type of the elements of the list being processed, and R be the type of the initial, intermediate, and final results):

- f is associative, and so is f', where $f'(x, y) = f(y, x)$, x is an arbitrary object of type `A`, and y is an arbitrary object of type `R`.
- If $f'(x, Z) = f(Z, x) = x$.

then we really don't care how what order the calls to f occur, or how many times we fold Z into the mix, since it's like an identity element.]]