

Enumerating the Strings of a Regular Expression

Jayadev Misra

8/29/2000

We develop an algorithm to enumerate the strings of a regular expression in increasing order, as defined below.

Let z be an alphabet whose symbols are totally-ordered. For finite strings s and t over z , we say that s is smaller than t , written $s < t$, if either (1) the length of s is smaller than that of t , or (2) the lengths of s and t are equal and s is lexicographically smaller than t , based on the ordering over z . Henceforth, all strings are finite.

Let r be a regular expression over z . It is required to enumerate the strings of r in increasing order (with respect to $<$). A regular expression r is defined as follows: (1) r denotes the empty set, (2) r contains only the null string, (3) r contains a string which is a single symbol of the alphabet, or for some regular expressions u and v , (4) $r = u \mid v$, (5) $r = u \cdot v$, or (6) $r = u^*$. For the first three cases, enumeration is straightforward. We discuss the remaining cases next.

1 Enumeration Problem

Let $enum$ be a function such that for a regular expression r , $enum(r)$ is the ordered list of strings denoted by r . For r of the form $u \mid v$, we apply $enum$ to both u and v and then merge the two lists to obtain $enum(r)$, i.e., using an infix operator $+++$ for function $merge$ that is yet to be defined,

$$enum(u \mid v) = (enum\ u) +++ (enum\ v)$$

Note: Henceforth, we use the Haskell convention that function application is left-associative, and it has the highest binding power. Therefore, parenthesis around the arguments can be eliminated in a number of cases.

For a regular expression of the form $u \cdot v$, we use a function $prod$, written as an infix operator $***$, to concatenate pairs of strings from the argument lists.

$$enum(u \cdot v) = (enum\ u) *** (enum\ v)$$

Finally, for a regular expression of the form u^* , we apply a function *closure*.

$$\text{enum}(u^*) = \text{closure} (\text{enum } u)$$

The remaining task is to define *merge*, *prod*, and *closure*, which we do next. Functions *merge* and *prod* have two arguments each; each argument is an ordered list of strings and the result is an ordered list of strings. Function *closure* has a single argument which is an ordered list of strings and its result is an ordered list of strings.

2 *merge*, *prod*, *closure*

For comparisons of strings, we define a metric on a string as follows. Two strings can then be compared by lexicographically comparing the corresponding metrics.

$$\text{metric } x = (\text{length } x, x)$$

2.1 Definition of *merge*

Function *merge* has two ordered lists of strings, **xs** and **ys**, as arguments; its result is an ordered list obtained by merging **xs** and **ys**. There are no duplicates in the resulting list provided each of **xs** and **ys** is duplicate-free. The following algorithm, written in Haskell, is self-explanatory. (The Haskell notation **xs@(x:xt)** denotes a list **xs** whose first element is **x** and the remaining part is the list **xt**.)

```
(+++) :: [String] -> [String] -> [String]
[] +++ ys = ys
xs +++ [] = xs
xs@(x:xt) +++ ys@(y:yt) =
  case compare (metric x) (metric y) of
    LT -> x : (xt +++ ys)
    EQ -> x : (xt +++ yt)
    GT -> y : (xs +++ yt)
```

2.2 Definition of *prod*

Function *prod* has two ordered lists of strings, **xs** and **ys**, as arguments; its result is an ordered list obtained from the concatenations of pairs of strings, from **xs** with those from **ys**. There are no duplicates in the resulting list provided each of the argument lists is duplicate-free. The following algorithm, written in Haskell, is mostly self-explanatory.

```
(***) :: [String] -> [String] -> [String]
[] *** _ = []
_ *** [] = []
xs@(x:xt) *** ys@(y:yt)
    = (x ++ y): ((xt *** [y]) +++ (xs *** yt))
```

The last case needs some explanation. The two argument lists are $(x:xt)$ and $(y:yt)$; the arguments are also denoted by **xs** and **ys**, respectively. Since **xs** and **ys** are assumed to be ordered, **x** is the smallest string in **xs** and **y** in **ys**. Hence, $x ++ y$ is the smallest string in **xs *** ys**, from the monotonicity of concatenation in both of its arguments. The remaining strings in the result list are of two kinds: (1) those that use **y** as part of a concatenation, i.e., the strings in $xt *** [y]$ and (2) those that do not, namely **xs *** yt**. The terms $xt *** [y]$ and **xs *** yt** are ordered lists of strings; therefore, merging them results in a single ordered list.

2.3 Definition of *closure*

Function *closure* applied to an ordered list of strings, **xs**, creates finite concatenations of all the strings in **xs**. Consider three cases (in the following "" is the null string): (1) $[]^* = [\""]$, (2) $(\"\": \text{xs})^* = \text{xs}^*$, and (3) if **xs** has no null string, then $\text{xs}^* = \text{xs}^*$: $(\text{xs} *** \text{xs}^*)$. Note that the order in which the three cases are written matters; in the last case, **xs** is a list which does not contain the null string.

```
closure :: [String] -> [String]
closure []           = [\""]
closure (\"\": xt) = closure xt
closure xs          = (\"\": (xs *** (closure xs)))
```

3 The Complete Haskell program

The following program, written in Haskell, uses exactly the same code except for minor syntactic differences. First, we define regular expressions, `Rexp`, over a type `z`. We use infix operators `:|` and `::` for concatenation and alternation.

```
-- Define concatenation and alternation to be associative.
-- concatenation has higher binding power than alternation.

infixr 5 :|
infixr 6 ::

data Rexp z =
  Phi           -- empty language
  | Nil         -- language containing null string
  | Single z    -- symbol from the alphabet
  | (Rexp z) :| (Rexp z) -- alternation
  | (Rexp z) :: (Rexp z) -- concatenation
  | Star (Rexp z) -- Kleene closure
```

The result of function `enum` is an ordered list of strings of a regular expression.

```
enum :: (Ord z, Show z) => Rexp z -> [String]
enum z = case z of
  Phi      -> []           -- empty language
  Nil      -> ["" ]       -- language containing null string only
  Single x -> [show x]    -- convert x to string
  x :| y   -> (enum x) +++ (enum y)
  x :: y   -> (enum x) *** (enum y)
  Star x   -> closure (enum x)
```

The following functions —*merge*, *prod*, and *closure*— are as given before.

```
(+++) :: [String] -> [String] -> [String]
[] +++ ys = ys
xs +++ [] = xs
xs@(x:xt) +++ ys@(y:yt) =
```

```

    case compare (metric x) (metric y) of
    LT -> x : (xt +++ ys)
    EQ -> x : (xt +++ yt)
    GT -> y : (xs +++ yt)

(***) :: [String] -> [String] -> [String]
[] *** _ = []
_ *** [] = []
xs@(x:xt) *** ys@(y:yt)
    = (x ++ y): ((xt *** [y]) +++ (xs *** yt))

closure :: [String] -> [String]
closure[] = []
closure ("": xt) = closure xt
closure xs = (""): (xs *** (closure xs))

```

Acknowledgment The Tuesday Afternoon Club, under the guidance of Edsger W. Dijkstra, read and commented on an earlier draft of this manuscript. Pete Manolios developed a program in ACL-2 based on the earlier draft. The current version is helped by several features of Haskell, the type definition mechanism and lazy evaluation, in particular. I am indebted to Doug McIlroy who provided me with the *merge* and *prod* functions, and explored a number of other solutions with me. Ham Richards has helped me with the intricacies of Haskell.