JFP **14** (5): 503–518, September 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796803004982 Printed in the United Kingdom

FUNCTIONAL PEARL
Enumerating the strings of regular languages

M. DOUGLAS McILROY
Dartmouth College, Hanover, NH 03755, USA
(*e-mail:* `doug@cs.dartmouth.edu`)

Abstract

Haskell code is developed for two ways to list the strings of the language defined by a regular expression: directly by set operations and indirectly by converting to and simulating an equivalent automaton. The exercise illustrates techniques for dealing with infinite ordered domains and leads to an effective standard form for nondeterministic finite automata.

Lazy languages are well suited to sequence generation because of the ease with which they handle infinite sequences. The problem of enumerating the strings of a regular language was proposed by Jay Misra as a lovely case in point, simple to state, yet tricky to solve. This paper develops two concise and radically different solutions in Haskell (Peterson *et al*., 1998), one direct and one indirect. Both approaches exploit Haskell's outstanding fitness for structural induction.

The direct approach interprets regular expressions as recipes for building sets. Here the main concern is programming the primitives for combining sets: union, cross-product, and closure under a binary operation. The primitives apply to any well ordered domain.

The indirect approach detours through automata, first constructing an equivalent nondeterministic finite automaton, then tracing execution paths of the automaton in breadth-first order. The automata have an unusual shape: besides a single final state, there is exactly one state per symbol in the regular expression and that state accepts only the one symbol. The set operations developed for the first solution are used again, this time for combining sets of states rather than sets of strings.

Working Haskell code is available electronically (McIlroy, 2003).

1 The problem

Devise a program to enumerate the distinct strings of the regular language denoted by a given regular expression. The resulting list should be ordered by string length and lexicographically within each length. Parsing is not at issue; regular expressions are taken to be already parsed data structures.

Table 1. *Meaning of regular expressions*

e	$L(e)$
$\emptyset$	empty set
ϵ	singleton set of the empty string
a	singleton set of the one-symbol string a
(e)	$L(e)$
e^*	Kleene closure: least fixed point of $k = \epsilon \vert ek$
e_1e_2	catenation: $\{x_1x_2 \vert x_1 \in L(e_1), x_2 \in L(e_2)\}$
$e_1 \vert e_2$	alternation: $L(e_1) \cup L(e_2)$
Conventions	
$e\ e_1\ e_2$	regular expressions
a	symbol of alphabet A
$\epsilon\ \emptyset$ ( ) * |	metacharacters, not symbols

1.1 Terminology

A *regular expression* denotes a language (or set) of strings over an ordered alphabet A. Table 1 defines how a regular expression e and its language $L(e)$ may be constructed. The operators (Kleene closure, catenation, alternation) are listed in decreasing order of precedence, subject as usual to explicit grouping by parentheses.

The catenation of two languages is the set of pairwise catenations of strings from each. The Kleene closure of a language is the set of strings made by catenating zero or more (not necessarily distinct) strings from the language.

Phrases such as ‘enumerating the strings of a language’ will usually be shortened to ‘enumerating a language’, and ‘enumerating the strings of the language defined by …’ to ‘enumerating the language of …’.

Two languages will be used as running examples.

Example 1, sandwich language
Regular expression `ab*a` denotes possibly empty strings of `b`’s sandwiched between single `a`’s:

```
["aa", "aba", "abba", "abbba", ...]
```

Example 2, even-a language
Regular expression `(ab*a|b)*` denotes strings that contain an even number of `a`’s and any number of `b`’s:

```
["aa", "aab", "aba", "baa", "aaaa", "aabb", "abab",
 "abba", "baab", "baba", "bbaa", "bbbb", ...]
```

2 Length-ordered lists

Length-ordered strings, LOS, are defined as a special case of a polymorphic *length-ordered list* data type, LOL.

```
data LOL a = LOL [a] deriving Eq
instance Ord a => Ord (LOL a) where
    LOL x <= LOL y = (length x, x) <= (length y, y)
type LOS = LOL Char
```

Equality tests are automatically lifted from lists to length-ordered lists by the `deriving` clause. Inequality tests are implemented by placing length ordered lists over an ordered type a in type class Ord and stating that comparison is done first on length, then content.

To define catenation of length-ordered lists, they are placed in class MonadPlus, to which the ++ operator belongs, and the constructor LOL is declared to distribute across catenation. As a technical detail, membership in MonadPlus requires membership in superclasses MonadZero and Monad.

```
instance MonadPlus LOL where
    (LOL x) ++ (LOL y) = LOL (x++y)
instance MonadZero LOL
instance Monad LOL
```

3 Set operations

According to the usual head-tail strategy for stream processing, operators to calculate new ordered sets from old ones will explicitly calculate the head (least) element of an output and recur to calculate the tail (greater) elements. These operators apply to strictly ordered lists of distinct elements.

3.1 Union

Set union is denoted by infix \/, left associative, with the same precedence (6) as addition. The operation is carried out by a conventional merge, using Haskell's 3-way comparison function.

```
infixl 6 \/
(\/) :: Ord a => [a] -> [a] -> [a]
[] \/ ys = ys
xs \/ [] = xs
xs@(x:xt) \/ ys@(y:yt) = case compare x y of
    LT -> x : xt\/ys
    EQ -> x : xt\/yt
    GT -> y : xs\/yt
```

(According to custom, sequence variables are suffixed with s. When both a sequence and its tail are named, the latter is given suffix t.) Other binary operations on sets, such as intersection, difference and symmetric difference, would be programmed similarly.

Example 3
These expressions all evaluate to `True`.

```
[1,3,4] \/ [1,2,4,7] == [1,2,3,4,7]
["a", "aab"] \/ ["b", "bb"] == ["a", "aab", "b", "bb"]
[LOL "a", LOL "aab"] \/ [LOL "b", LOL "bb"]
    == [LOL "a", LOL "b", LOL "bb", LOL "aab"]
```

Example 4
The expression

```
[0,2..] \/ [1,3..] == [1..]
```

though true, involves infinitely many comparisons; its evaluation will not terminate.

3.2 Cross product and set catenation

To catenate sets, we must generate an ordered list of strings xy for all pairs (x, y) in the cross product of two sets. The least string of the result is the catenation of the least strings of the two sets. Catenation of length-ordered lists is monotone in both arguments: $x < x'$ implies $xy < x'y$ for all y and $yx < yx'$ for all finite y. Other familiar binary functions have the same property: addition of nonnegative integers, multiplication of positive integers, and pairing of nonnegative integers in the Cantor order, where pairs (i, j) are ordered first on $i + j$ and then on i. It should be useful, then, to make a polymorphic cross-product generator with a functional argument:

```
xprod :: (Ord a, Ord b, Ord c) => (a->b->c) -> [a] -> [b] -> [c]
```

Example 5
These expressions evaluate to `True`:

```
xprod (+) [1,3,5] [2,4,5] == [3,5,6,7,8,9,10]
xprod (++) ["","a"] ["ab", "b"] == ["ab","aab","b"]
xprod (++) [LOL "", LOL "a"] [LOL "b", LOL "ab"]
    == [LOL "b", LOL "ab", LOL "aab"]
xprod (^) [2,3,4] [1,2] == [2,3,4,9,16]
(xprod pair [1,2] [1,2] == [(1,1),(1,2),(2,1),(2,2)])
    where pair x y = (x,y)]
```

The function `xprod` can work, in the sense that every element of the cross product eventually appears, only if the output domain is *well ordered,* i.e. has no element with an infinite number of predecessors. Length-ordered lists are well ordered.

Example 6
The last expression in Example 5 does not generalize to infinite lists. The expression

```
(xprod pair [1..] [1..]) where pair x y = (x,y)
```

fails to produce a full mathematical cross product, because Haskell's standard (lexicographic) ordering on pairs is not a well ordering. The expression evaluates to

```
[(1,1),(1,2),(1,3), ...]
```

which contains no pair with first member greater than 1.

The basic pattern of `xprod` applied to nonempty sets must be

```
xprod f (x:xt) (y:yt) = f x y : tail
```

The `tail` may be decomposed into two sets of pairings: (1) `x` paired with everything in `yt`, and (2) everything in `xt` paired with everything in `(y:yt)`. The two sets may be constructed recursively and then unioned to give the tail.

```
xprod _ [] _ = []
xprod _ _ [] = []
xprod f (x:xt) ys@(y:yt) =
    (f x y) : (xprod f [x] yt) \/ (xprod f xt ys)
```

Catenation of regular languages, represented as length-ordered lists of strings, is a specialization.

```
cat :: [LOS] -> [LOS] -> [LOS]
cat = xprod (++)
```

3.3 Closure

The Kleene closure, x^*, of a set x of strings is the closure of x under catenation, i.e. x^* is the least fixed point k of $k = \epsilon \mid xk$. Again we generalize. The set x of strings becomes a set from a well ordered domain; catenation becomes a semigroup operation f on the domain, monotone in both arguments; and ϵ becomes the identity element z for operation f. Moreover, z must be the least element of the domain.

```
closure :: Ord a => (a->a->a) -> a -> [a] -> [a]
closure f z [] = [z]
closure f z xs@(x:xt) = if x==z
    then closure f z xt
    else z : xprod f xs (closure f z xs)
```

This definition of `closure` works, despite the seemingly circular call in the `else` clause, because the cons constructor `(:)` assures a lag; each new element in the result depends only on previously computed elements. The critical `then` clause deletes the identity element from `xs`, thereby preventing output elements from reappearing by combination with the identity. The deletion causes nothing to be omitted from the cross product.

Kleene closure, `clo`, is a specialization, in which the operation is catenation and the identity element is the (length-ordered) empty string.

```
clo :: [LOS] -> [LOS]
clo = closure (++) (LOL "")
```

```
data Rexp = Nil                 -- empty language
          | Eps                 -- empty string
          | Sym Char            -- symbol of the alphabet
          | Clo Rexp            -- Kleene closure
          | Cat Rexp Rexp       -- catenation
          | Alt Rexp Rexp       -- alternation

enumR :: Rexp -> [String]
enumR r = [x | (LOL x) <- enumR' r]

enumR' :: Rexp -> [LOS]
enumR' Nil       = []
enumR' Eps       = [LOL ""]
enumR' (Sym a)   = [LOL [a]]
enumR' (Clo x)   = clo (enumR' x)
enumR' (Cat x y) = cat (enumR' x) (enumR' y)
enumR' (Alt x y) = alt (enumR' x) (enumR' y)

alt :: [LOS] -> [LOS] -> [LOS]
alt = (\/)
```

Fig. 1. Direct enumeration of regular languages.

4 Direct enumeration

Given the set operations, it is easy to enumerate a language directly from its defining regular expression. Table 1 maps directly into the data definition in Figure 1, where a straightforward program `enumR'` constructs a list of length-ordered strings, using set operations. A wrapper program, `enumR`, strips away the `LOL` constructors to give a list of plain strings.

Example 7
The sandwich and even-a languages, `ab*a` and `(ab*a|b)*`, are defined by

```
a = Sym 'a'
b = Sym 'b'
sandwich = Cat a (Cat (Clo b) a)
even_a = Clo (Alt sandwich b)
```

and the even-a language is enumerated by

```
enumR even_a
```

5 A path through automata

Automata theory offers a radically different approach to enumerating regular languages. A Nondeterministic Finite Automaton (NFA) is a convenient equivalent to a regular expression, with a state count roughly linear in the length of the expression. (Equivalent deterministic automata can be exponentially larger.) We shall convert regular expressions to nondeterministic automata (Perrin, 1990) of a particularly simple form, with these properties:

- There is one final state.
- There is also one state for each occurrence of a symbol in the regular expression.
- One or more states are start states, unless the automaton is empty.
- Each nonfinal state accepts just one symbol.
- There are no epsilon moves; the input tape moves with every state change.

Example 8
The automaton for the even-a language `(ab*a|b)*` has 5 states: a final state and one state for each occurrence of symbols `a` and `b`.

An automaton is known by its start states. A state is fully described by a distinguishing identifier, the symbol it accepts, and the states it moves to on that symbol. 'The states it moves to' may equally well be thought of as 'the new automaton it becomes'. We take state identifiers to be nonnegative integers, with 0 identifying the final state.

```
type NFA = [State]
data State = State Ident Char NFA
type Ident = Int
```

Example 9
The sandwich language `ab*a` is defined by automaton `g`, whose only start state is `g3`.

```
g0 = State 0 '~' []
g1 = State 1 'a' [g0]
g2 = State 2 'b' [g1,g2]
g3 = State 3 'a' [g1,g2]
g = [g3]
```

States in `g` are numbered from right to left in the regular expression. The final state's symbol, `~`, is a dummy, unusable because there are no moves from that state.

Example 10
The even-a language `(ab*a|b)*` is defined by automaton `h`, with three start states.

```
h0 = State 0 '~' []
h1 = State 1 'b' [h4,h1,h0]
h2 = State 2 'a' [h4,h1,h0]
h3 = State 3 'b' [h2,h3]
h4 = State 4 'a' [h2,h3]
h = [h4,h1,h0]
```

5.1 Terminology

The pedantic distinction between a regular expression and its equivalent automaton may be dropped when the meaning is clear from context.

An automaton e whose language $L(e)$ contains ϵ is called *bypassable*.

$b(e)$ is a predicate, true if and only if e is bypassable. $S(e)$ denotes the set of start states of automaton e. $F(e)$ denotes the set of *first states,* start states of e that are distinct from the final state. The language of e started in $F(e)$ is $L(e)-\epsilon$. $D(e)$ denotes *destination states,* states that are outside automaton e and are reached by notional epsilon moves from the final state of e. The notional epsilon moves are subsumed by replacing each move to the final state with a set of moves to the destination states. The final state of any subautomaton is also notional; it is never represented in any data structure.

A set of *constituent equations* relates the attributes $b(e)$, $S(e)$, $F(e)$ and $D(e)$ of a composed automaton e and of any immediate components, e_1 or e_2.

Destination states of a bypassable automaton also appear among the start states according to the constituent equations

$$\begin{aligned} S(e) &= F(e) \cup D(e), && \text{if } b(e) \text{ is true} \\ S(e) &= F(e), && \text{if } b(e) \text{ is false} \end{aligned}$$

Example 11
Automaton `h` (Example 10) has two first states, `h4` for the first `a` in `(ab*a|b)*` and `h1` for the second `b`. Being a closure, which can match the empty string, the body of the automaton is bypassable. Thus the start states include the first states and the sole destination state, final state `h0`.

5.2 NFA construction

We shall define a function `r2n` to convert a regular expression to a nondeterministic automaton.

```
r2n :: Rexp -> NFA
```

The main work of conversion, though, will be done by an auxiliary function `r2n'`, which constructs automata for subexpressions and connects them according to the recipes below. The parameters of `r2n'` are a subexpression (type `Rexp`), the least identifier (`Ident`) available for any states that may have to be created, and the destination states (`NFA`). The return value comprises the first states (`NFA`), the next available identifier (`Ident`) for use in further subautomata, and a bypass flag (`Bool`), true if and only if the newly constructed automaton is bypassable.

```
r2n' :: Rexp -> Ident -> NFA -> (NFA,Ident,Bool)
```

Example 12
Consider the subexpression that consists of the first `b` in `(ab*a|b)*`. The corresponding subautomaton, `h3` in Example 10, is built by the code fragment

```
(fs,n,bf) = r2n' (Sym 'b') 3 ([h2]\/fs)
```

which is invoked in the course of constructing the automaton for the immediately enclosing subexpression `b*`. State identifier `3` is the first identifier that the new subautomaton can use. State `h2`, which accepts the second `a` in `(ab*a|b)*`, is the only destination state outside of the surrounding `b*`. The full destination argument, `([h2]\/fs)`, says the subautomaton for `b` in context may also loop back to its own first states `fs`; lazy evaluation closes the loop. The new available-identifier value, `n`, will be 4 because the subautomaton uses one state, state 3. The bypass flag `bf` will be `False` because `b` cannot match the empty string.

The top-level conversion program, `r2n`, constructs the final state `(State 0 '~' [])`. and calls `r2n'` to construct the rest of automaton `r`. State `0` is the sole member of the automaton's destination states `ds`, and `1` is the next available state identifier. The start states returned by `r2n` include the first states and, conditionally, the destination states (the singleton final state), as determined by a bypass function `bp`, depending on the bypass flag `b`.

```
r2n r = let {
    ds = [State 0 '~' []];
    (fs, _, b) = r2n' r 1 ds
    } in fs \/ (bp b ds)

bp :: Bool -> NFA -> NFA
bp True ds = ds
bp False _ = []
```

Since operator `\/` works only on ordered sets, we must place an order on states. Though any ordering, such as ordering by identifier, might do, we shall see later that ordering by state symbol is more convenient for the task at hand.

5.2.1 Primitive automata

The constructions for primitive regular expressions e are straightforward. Each is described below, together with constituent equations and Haskell code.

If e is $\emptyset$, the automaton is nugatory. It has no states and is not bypassable.

$$F(e) = \emptyset$$
$$b(e) = \text{false}$$

```
r2n' Nil n _ = ([], n, False)
```

If e is ϵ, the automaton also has no states; it is a pure bypass.

$$F(e) = \emptyset$$
$$b(e) = \text{true}$$

```
r2n' Eps n _ = ([], n, True)
```

If e is `(Sym c)`, the automaton has one state, which accepts symbol `c`; it is not bypassable. Its identifier is `n`; the next available identifier is `n+1`. The state's moves go to the destination states `ds`.

$$F(e) = \{\text{state n}\}$$
$$b(e) = \text{true}$$

```
r2n' (Sym c) n ds = ([State n c ds], n+1, False)
```

5.2.2 Composed automata

In an alternation, $e = e_1 \mid e_2$, the destination states of e_1 and e_2 are the destination states of e. The first states of e are the union of the first states of e_1 and e_2. The alternation is bypassable if either e_1 or e_2 is bypassable.

$$\begin{gathered} D(e_1) = D(e_2) = D(e) \\ F(e) = F(e_1) \cup F(e_2) \\ b(e) = b(e_1) \vee b(e_2) \end{gathered}$$

```
r2n' (Alt x y) n ds = let {
    (fs, n', b) = r2n' y n ds;
    (fs', n'', b') = r2n' x n' ds;
    } in (fs\/fs', n'', b||b')
```

In a catenation, $e = e_1e_2$, the destination states of e_2 are the destination states of e. The destination states of e_1 are the start states of e_2. The first states of the catenation are the first states of e_1 plus, if e_1 is bypassable, the first states of e_2. The catenation is bypassable if both e_1 and e_2 are bypassable.

$$\begin{gathered} D(e_2) = D(e) \\ D(e_1) = S(e_2) \\ b(e) = b(e_1) \wedge b(e_2) \\ \begin{aligned} F(e) = F(e_1) \cup F(e_2), &\quad \text{if } b(e_1) \text{ is true} \\ F(e) = F(e_1), &\quad \text{if } b(e_1) \text{ is false} \end{aligned} \end{gathered}$$

```
r2n' (Cat x y) n ds = let {
    (fs, n', b) = r2n' y n ds;
    (fs', n'', b') = r2n' x n' (fs\/(bp b ds));
    } in (fs'\/(bp b' fs), n'', b&&b')
```

The first states of a closure, $e = e_1^*$, are the first states of e_1. The destination states of e_1 are the union of the first states of e_1 and the destination states of e. The closure is bypassable.

$$\begin{gathered} F(e) = F(e_1) \\ D(e_1) = D(e) \cup F(e_1) \\ b(e) = \text{true} \end{gathered}$$

```
r2n' (Clo x) n ds = let {
    (fs, n', _) = r2n' x n (fs\/ds)
    } in (fs, n', True)
```

Example 13
The automata in Examples 9 and 10 are exactly the automata that `r2n` builds for the regular expressions of Example 7.

```
g = r2n sandwich
h = r2n even_a
```

5.2.3 Sufficiency of lazy evaluation

Inspection of the constituent equations reveals that the following computation schedule imposes an order of evaluation wherein the left side of each equation may be calculated from the right. The existence of a feasible schedule assures us that lazy evaluation of `r2n'` will solve the constituent equations.

1. Initialize $D(e)$ at the root of the regular expression tree, from `r2n`.
2. Initialize $b(.)$ and F(.) at the leaves, from formulas for primitive automata.
3. Compute $b(.)$ by a postorder traversal of the regular expression tree.
4. Compute $F(.)$ by a postorder traversal.
5. Compute $D(.)$ and $S(.)$ by a preorder traversal that computes $D(e)$ before $S(e)$ and visits e_2 before e_1 whenever both exist.

5.2.4 Complexity

Let a regular expression r contain s symbols ($s + 1$ states); let its parse tree contain n nodes; and let time be measured in Haskell interpreter steps, exclusive of compilation and garbage collection. Then the set operations in `r2n'` take worst-case time $O(s + 1)$ per node for which they are called. Since `r2n'` is called once per node, the construction takes time at most $O((s + 1)n)$, or $O(n^2)$ in view of the fact that $s \leqslant n$. This complexity is asymptotically optimal, witness the family of regular expressions

$$a_1^* a_2^* \dots a_n^*,$$

where the a_i are distinct. Minimal automata for these expressions have move tables with $O(n^2)$ entries.

5.3 Enumerating the language of an NFA

To enumerate the language of an automaton, we simulate its operation. Each step of the simulation treats the automaton's action in a set of states to which the automaton has been driven by a distinct string. The history to that point is summarized in a *word* that tells the string and the current states.

```
type Word = (String,NFA)
```

In each state the automaton accepts one symbol and moves to another set of states. Since different states may have different symbols, each word in general gives rise to multiple words, the strings of which differ in their last symbols. Thus the words form a tree rooted at a start word (string is empty, current states are the start states). A word's depth in the tree is the same as the length of its string. If a word's states include the final state (0), the word's string is accepted by the automaton.

```
accept :: NFA -> Bool
accept ds = 0 ‘elem‘ [i | (State i _ _) <- ds]
```

To produce strings ordered by length, we must walk the tree of words breadth first. Thus we are led to keep a queue `ws` of words.

```
visit :: [Word] -> [String]
visit [] = []
visit ((x,ds):ws) = let { xs = visit (ws ++ ...)
    } in if accept ds then x:xs else xs
```

The current word `(x,ds)` is removed from the queue and a set of successor words `(...)` is constructed and appended to the queue. If the current states indicate acceptance, the word's string is placed on the output list. The rest of the list, `xs`, is computed by recursively visiting other words in the queue.

To avoid getting duplicate strings among the successor words, we group the moves from all the states in `ds` by symbol. For convenience we represent a group by a (possibly fictitious) state that accepts the symbol and has a unioned list of moves. The code for the grouping function

```
grp :: NFA -> NFA
```

will be given later. The set of successors to word `(x,ds)` is

```
[(x++[c],ds') | (State _ c ds') <- grp ds]
```

Whence the completed code for `visit` is

```
visit [] = []
visit ((x,ds):ws) = let { xs = visit (ws ++
        [(x++[c],ds') | (State _ c ds') <- grp ds])
    } in if accept ds then x:xs else xs
```

If the grouped states are ordered by symbol, words with strings `x++[c]` will be added to the queue in lexicographic order.

To facilitate grouping, it is convenient to keep lists of states ordered by symbol, resolved when necessary by identifier. That ordering is defined by

```
instance Eq State where
    (State i _ _) == (State i' _ _) = i==i'
instance Ord State where
    (State i c _) <= (State i' c' _) = (c,i) <= (c',i')
```

Now, to enumerate the language of an automaton known by its start states `starts`, we initialize the visiting process at the root of the tree, with the automaton setting out from its start states to consider and extend the empty word.

```
enumA :: NFA -> [String]
enumA starts = visit [("",starts)]
```

Words will be visited in length-first order. To prove this, observe that if words of length n occur consecutively in the queue (as holds trivially for $n = 0$), then their successors of length $n + 1$ will occur consecutively further on in the queue. Moreover, if all words x of length n are lexicographically ordered (again trivially true for $n = 0$), the successor strings `x++[c]` will be, too, because the c's come in order.

We must still provide code for grp, which represents the union of moves of states that have the same symbol as the moves of fictitious states, all harmlessly given the same identifier, −1. Every list of states, having been constructed by set operations, is necessarily ordered, and states are ordered by symbol. Thus grp need only run along the list of states, consolidating adjacent states that have a common symbol c.

```
grp :: NFA -> NFA
grp (m@(State _ c ds) : ms@((State _ c' ds'):mt)) =
    if c==c' then grp ((State (-1) c (ds\/ds')):mt)
    else m : grp ms
grp ms = ms                   -- 0- and 1-element lists
```

While the foregoing code is complete, in the sense that it does eventually list each string of a language, it can run forever considering dead-end 'prefixes' for an empty language denoted by a regular expression such as $a^*\emptyset$. This bad behavior may be forestalled by preprocessing to eliminate $\emptyset$ from all but the top level of a regular expression. A function to do so, deNil, is included in the working code (McIlroy, 2003). A robust enumerator enumRA of the language of a regular expression may be composed from deNil, r2n and enumA.

```
deNil :: Rexp -> Rexp
enumRA :: Rexp -> [String]
enumRA = enumA . r2n . deNil
```

Example 14
The Haskell expression

```
enumRA sandwich
```

enumerates the sandwich language (Example 7).

5.4 Testing

A good test of both of the very different programs enumR and enumRA is afforded by two-version programming. Although Haskell cannot confirm equality of infinite lists, it can check arbitrary initial segments. Agreement on a long initial stretch is good evidence for the correctness of the two enumerators.[1]

To look for pathological cases, the comparative check was run on exhaustive enumerations of expressions of limited size, using the Hugs(1998) interpreter. The most extensive test checked thirty strings for each of the 182,712 expressions free of $\emptyset$ and ϵ on a two-letter alphabet, with operators nested at most three deep. This test, including the generation of regular expressions, used one billion reductions, two billion cells, 23000 garbage collections, and 30 CPU minutes on a 400 MHz

[1] A plausible, but usually wildly pessimistic, estimate of how far to check follows from setting the goal of exercising every distinct superstate (set of current states) of the NFA. As the number of superstates is at most 2^{s+1}, where s is the number of symbols in the regular expression, a correct implementation is sure to visit every superstate in the course of listing strings up to length 2^{s+1}.

Pentium with Hugs's default 100K workspace. The exhaustive lists of expressions with operator nesting depth at most d were created by specifying an ordering and using set operations.

```
rexprs 0 = [Sym 'a', Sym 'b']
rexprs d = let rs = rexprs (d-1) in rs \/
    xprod Cat rs rs \/ xprod Alt rs rs \/ map Clo rs
```

Another test, of the 112, 416 regular expressions having eight or fewer `Rexp` nodes, on two alphabetic symbols plus $\emptyset$ and ϵ, used 800 million reductions, 1.5 billion cells, 20000 garbage collections, and 25 CPU minutes.

6 Discussion

6.1 Efficiency

The direct enumerator `enumR` can waste time generating every parsing of a string and discarding all but one. Moreover, the time per comparison to identify duplicates grows with string length.

Example 16
Length-n strings in the language $\mathtt{a}^*\mathtt{a}^*$ have $n+1$ different parsings: $\mathtt{a}^i\mathtt{a}^{n-i}$, for $i = 0..n$. The direct enumerator makes $n+1$ copies of `a` and rejects all but one as duplicates. By catenating k instances of $\mathtt{a}^*$ we can make the number of comparisons per string of length n grow as n^{k-1}.

By contrast, an automaton generates each word just once. It may be shown that an automaton built from a $\emptyset$-free regular expression considers $O(m)$ words in generating the first m strings of the language, and does a bounded amount of work per word considered. Thus each such automaton does work asymptotically proportional to the number of language strings enumerated. Still, the work per string can be large, depending on the regular expression. For some regular expressions direct generation is faster.

6.2 Fictitious states

The fictitious states constructed in the course of visiting a word are closely related to an equivalent deterministic automaton. A word's current-state set, or *superstate*, may be identified with a state of the deterministic automaton. The fictitious states describe superstates and their transitions. By caching distinct fictitious states, one could build the equivalent deterministic automaton in the course of simulating the NFA.[2]

[2] This hypothetical way to build a deterministic automaton echos the eminently practical lazy construction strategy pioneered by Aho in the Unix pattern-matching program "egrep": construct a state of a deterministic automaton only when a string that drives the automaton to that state is met.

6.3 The automata

Recognizers based on nondeterministic automata in the one-state-per-symbol form are extremely simple to simulate. The execution cycle is

1. Check whether any current state is final.
2. Replace the set of current states by the union of next states for every current state that accepts the next input character.
3. Advance the input.

The construction method descends from Ken Thompson's (1968) classic construction. The present method differs from the earlier construction and derivatives thereof (Aho *et al*., 1986; Thompson, 2000) in dispensing with epsilon moves. Epsilon moves do not appear as either transitory or permanent artifacts of the construction process. There is no separate transitive-closure calculation to remove them, and no capability to simulate them. Only the tiny function `bp` remains as a Cheshire grin of epsilons past.

6.4 Paean to Haskell

The art of handling regular expressions and automata is ancient; I have simply dressed the subject in modern garb. Functional programming in general (Harper, 1999), and Haskell in particular (Thompson, 2000), offer eminently suitable fabric.

In developing the automata-based code, I set out guided by the classic models (Thompson, 1968; Aho *et al*., 1986). Then the Haskell formulation revealed an opportunity: lazy evaluation allowed separate states that represented composed subexpressions and associated epsilon moves to be elided prospectively.

Having found the Haskell version, I could now write the program comfortably in most any language. But if the exercise had begun in a traditional language, the final neat model would not have been perceived.

Acknowledgements

This work reflects the influence of members of IFIP Working Group 2.3, Programming Methodology. In particular Jay Misra proposed the problem, provided the elegant one-line definition of length order, and supplied the running examples. Bill McKeeman broke out of the box with an automaton-based solution in Matlab, which inspired `enumA`. Discussion and solutions in various languages by other members of the working group have shed further light on the problem. I am also grateful for unstinting advice from Phil Wadler about Haskell and Al Aho about automata.

References

Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers: Principles, Techniques and Tools.* Addison Wesley.

Harper, R. (1999) Proof-directed debugging. *J. Functional Program.* **9**, 463–469.

Hugs 1.4 (1998) University of Nottingham and Yale University. `http://www.haskell.org`.

Karczmarczuk, J. (1997) Generating power of lazy semantics. *Theor. Comput. Sci.* **187**, 203–219.

McIlroy, M. D. (2003) Enumerating the strings of regular languages. Accompanying online material, *J. Functional Program.* `http://www.dcs.glasgow.ac.uk/jfp/bibliography/author.html`.

Perrin, D. (1990) Finite automata. In: van Leeuwen, J. (editor), *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*. Elsevier.

Peterson, J. *et al.* (1998) Report on the Programming Language Haskell, a Non-strict, Purely Functional Language. `http://www.haskell.org`.

Thompson, K. (1968) Regular expression search algorithm. *Comm ACM,* **11**, 419–422.

Thompson, S. (2000) Regular Expressions and Automata using Haskell. Technical report 5-00, Computing Laboratory, University of Kent.