Regenerate: A Language Generator for Extended Regular Expressions

Gabriel Radanne, Peter Thiemann

Regenerate: A Language Generator for Extended Regular Expressions

GABRIEL RADANNE, University of Freiburg, Germany
PETER THIEMANN, University of Freiburg, Germany

Regular expressions are part of every programmer's toolbox. They are used for a wide variety of language-related tasks and there are many algorithms for manipulating them. In particular, matching algorithms that detect whether a word belong to the language described by a regular expression is both a well explored area, yet one that still receives frequent contributions. However, there is no satisfactory solution for testing such matchers, which would require generating positive as well as negative examples for the language.

We propose an algorithm to generate a language matched by a generalized regular expression with intersection and complement operators. The complement operator allows us to generate both positive and negative example words from a given regular expression. We implement our generator in Haskell and OCaml, and show that its performance is more than adequate for the purpose of testing.

Additional Key Words and Phrases: Regular expressions; parsing; formal languages; power series; Haskell; OCaml

1 INTRODUCTION

Regular languages are everywhere. Due to their apparent simplicity and their concise representability in the form of regular expressions, regular languages are used for many text processing applications, reaching from text editors [Thompson 1968] to extracting data from web pages.

Consequently, there are many algorithms and libraries that implement parsing for regular expressions. Some of them are based on Thompson's translation from regular expressions to nondeterministic finite automata and then apply the powerset construction to obtain a deterministic automaton. Others are based on Brzozowski's derivatives [Brzozowski 1964] and map a regular expression directly to a deterministic automaton. Antimirov's partial derivatives [Antimirov 1996] yield another transformation into a nondeterministic automaton. An implementation based on Glushkov automata has been proposed [Fischer et al. 2010] with decent performance. Cox's webpage [Cox 2007] gives a good overview of efficient implementations of regular expression search. It includes a discussion of his implementation of Google's RE2 [Cox 2010]. Current research still uncovers new efficient algorithms for matching subclasses of regular expressions [Groz and Maneth 2017].

Some of the algorithms for regular expression matching are rather intricate and the natural question arises how to test these algorithms. While there online repositories with reams of real life regular expressions [RegExLib [n. d.]], there are no satisfactory generators for test inputs. It is not too hard to come up with generators for strings that match a given regular expression, but that is only one side of the medal. On the other hand, the algorithm should reject strings that do not match the regular expression, so it is equally important to come up with strings that do **not** match.

This work presents generator algorithms for extended regular expressions that contain intersection and complement beyond the regular operators. The presence of the complement operator enables the algorithms to generate strings that certainly do not match a given (extended) regular expression.

Our implementations are useful in practice. They are guaranteed to be productive and produce total outputs. That is, a user can gauge the string size as well as the number of generated strings without risking partiality.

2018. Manuscript submitted to ACM

Even though the implementations are not tuned for efficiency they generate languages at a rate between $1.3 \cdot 10^3$ and $1.4 \cdot 10^6$ strings per second, for Haskell, and up to $3.6 \cdot 10^6$ strings per second, for OCaml. The generation rate depends on the density of the language.

The development of the generator is explained using both Haskell and OCaml and implementations for both languages will be available for artifact evaluation. The overall design makes significant use of laziness. This part of the design space is investigated and explained using Haskell in Sections 2 and 3. Fine tuning of the underlying data structures is investigated using OCaml in Section 4. Section 5 reports benchmarking results, Section 6 discusses related work, and Section 7 concludes.

We assume fluency with Haskell and OCaml throughout the paper. Some familiarity with formal languages is helpful, but not required as the paper contains all relevant definitions. Our notation for formal languages is borrowed from one of the classic textbooks on the topic [Hopcroft et al. 2003].

2 MOTIVATION

Suppose someone implemented a clever algorithm for regular expression matching. We want to use this implementation, but we also want to play safe and make sure it is largely bug free by subjecting it to extensive testing—verification is deemed to expensive. Such testing requires us to come up with test cases and implement a test oracle.

A test case consists of a regular expression `r` and an input string `s`. If `match` is the implementation of matching and `matchOracle` is the test oracle, then executing the test case means to execute `match r s` and check whether the result is correct by comparing it with `matchOracle r s`.

A popular way of conducting such a test is using the QuickCheck library [Claessen and Hughes 2000], which performs property-based random testing. Using QuickCheck, we would write a random generator for regular expressions and then use the random generator for strings to generate many inputs for a generated regular expression.

However, this approach has a catch. Depending on the language of the regular expression, the probability that a random string is a member of the language can be severely skewed. As an example, consider the language $L = (ab)^*$ over the alphabet $\Sigma = \{a, b\}$. Although L contains infinitely many words, the probability that a random word of length n is an element of L is

- 0 if n is odd and
- $\frac{1}{2^n}$ if n is even.

Thus, the probability p_n that a random word of length less than or equal to n is an element of L is very small:

$$p_n = \frac{\lfloor n/2 \rfloor}{2^{n+1} - 1} \leq \frac{n}{2^{n+2} - 2}$$

The probability $P(w \in L)$ of (uniformly) randomly selecting a word in L is zero in the limit.

Hence, there are two problems with testing the regular expression matcher.

(1) How do we know whether the test oracle is correct, short of verifying it?
(2) How do we ensure that relevant test cases are generated, given that the probability of randomly picking a word in the language is 0 or 1 for many regular languages?[1]

[1] Exercise for the interested reader:
(a) Come up with a regular language R such that $P(w \in R)$ is different from 0 or 1.
(b) Given a proper fraction m/n (that is $n > 0$ and $0 \leq m \leq n$) define a regular language R such that $P(w \in R) = m/n$.

r, s	$[\![\_]\!] =$		`data GRE sig`
::= **0**	empty	$\emptyset$	`= Zero`
| **1**	empty word	$\{\varepsilon\}$	`| One`
| $(a \in \Sigma)$	singleton	$\{a\}$	`| Atom sig`
| $r + s$	alternative	$[\![r]\!] \cup [\![s]\!]$	`| Or (GRE sig) (GRE sig)`
| $r \cdot s$	concatenation	$[\![r]\!] \cdot [\![s]\!]$	`| Dot (GRE sig) (GRE sig)`
| r^*	Kleene star	$[\![r]\!]^*$	`| Star (GRE sig)`
| $r \,\&\, s$	intersection	$[\![r]\!] \cap [\![s]\!]$	`| And (GRE sig) (GRE sig)`
| $\sim r$	complement	$\Sigma^* \setminus [\![r]\!]$	`| Not (GRE sig)`

Fig. 1. Generalized regular expressions

Wouldn't it be nice to have a systematic and obviously correct means of generating words **inside** of L and **outside** of L? Such a generation algorithm would obviate the need for an oracle and it would make sure that we can control the number of test inputs in the language and in the language's complement.

In the follwing we will tackle the slightly more general question of generating the language of a *generalized regular expression*, which subsumes the purpose of generating positive and negative sample words for testing.

2.1 Brief Intermezzo on Formal Languages

As customary, we write Σ^* for the set of finite words over alphabet Σ, which is defined by $\bigcup_{i=0}^{\infty} \Sigma^i$ where $\Sigma^0 = \{\varepsilon\}$ and $\Sigma^{i+1} = \Sigma \times \Sigma^i$.[2] The semantics of an expression, $[\![r]\!] \subseteq \Sigma^*$, is a set of words, which is also defined in Figure 1. It relies on standard definitions from the theory of formal languages. We write ε for the empty word and $u \cdot v$ for the concatenation of words $u, v \in \Sigma^*$. We write $|u|$ for the length of word u. Unless otherwise specified, we use $a, b, c, \ldots$ to range over Σ and $u, v, w, \ldots$ to range over Σ^*.

If $U, V \subseteq \Sigma^*$ are languages, then their concatenation (or product) is defined as $U \cdot V = \{u \cdot v \mid u \in U, v \in V\}$. We sometimes write $u \cdot V$ as an abbreviation for the product $\{u\} \cdot V$ with a singleton language. The Kleene closure of a language $U \subseteq \Sigma^*$ is defined as $U^* = \bigcup_{i=0}^{\infty} U^i$ where $U^0 = \{\varepsilon\}$ and $U^{i+i} = U \cdot U^i$.

A generalized regular expression (Figure 1) is an expression built from the regular operators empty set, empty word, singleton word consisting of a single letter a chosen from a finite alphabet Σ, alternative, concatenation, and Kleene closure. In addition, it may contain the operators intersection and complement. The extra operators do not add extra descriptive power as regular languages are closed under intersection and complement [Hopcroft et al. 2003], but generalized regular expressions can be much more concise.

2.2 Naive Approach

We start with a naive implementation of the mathematical definition in Figure 1. We define an alphabet by a list of `Char`. We represent words by elements of Haskell's `Data.Text.Text` datatype, abbreviated to `T.Text`. We represent a language as a lazy list of `Text`, as a language can be an infinite set. There are two further restrictions.

(1) The output of a generator should not contain repetitions because we would like to guarantee that test inputs are different from each others.

[2] Equivalently define Σ^* inductively as the smallest set such that $\varepsilon \in \Sigma^*$ and $\forall a \in \Sigma, \forall w \in \Sigma^*, aw \in \Sigma^*$

```haskell
module Examples.Naive where
import qualified Data.Text as T

union :: Lang -> Lang -> Lang
union lx ly = lx ++ ly

concatenate :: Lang -> Lang -> Lang
concatenate lx ly = [T.append wx wy | wx <- lx, wy <- ly ]

intersect :: Lang -> Lang -> Lang
intersect lx ly = [wx | wx <- lx, wx `elem` ly ]

star :: Lang -> Lang
star lx = concat lxi
  where
    lxi = [T.empty] : map (concatenate lx) lxi

complement :: Alphabet -> Lang -> Lang
complement sigma lx =
  undefined
```

Fig. 2. Partial implementation of the regular operators

(2) The output of a generator should not be partial because it would lead the test code to hang on a nonterminating input.

```haskell
import Data.Text as T

type Alphabet = [Char]
type Lang = [T.Text]

generate :: Alphabet -> GRE Char -> Lang
generate sigma r = gen r
  where
    gen Zero = []
    gen One  = [T.empty]
    gen (Atom t) = [T.singleton t]
    gen (Or r s) = union (gen r) (gen s)
    gen (Dot r s) = concatenate (gen r) (gen s)
    gen (Star r) = star (gen r)
    gen (And r s) = intersect (gen r) (gen s)
    gen (Not r) = complement sigma (gen r)
```

As a basis for further discussion, we exhibit a partial implementation in Figure 2. This implementation has a number of deficiencies.

union The output may contain duplicates. If `lx` is infinite, then no words from `ly` will ever be produced. This behavior violates the specification of set union because there may be elements in `ly` that never appear in `union lx ly`.
If we restricted ourselves to finite lists, then replacing `++` with `Data.List.union` would be an appropriate implementation, but its worst-case time complexity is quadratic.

concatenate The output may contain duplicates. If `ly` is infinite, then only the first word in `lx` contributes to the output.
For finite lists, an appropriate implementation would compose the raw product computation with `Data.List.nub` to remove duplicates. The worst-case complexity of `nub` is quadratic.

intersect The output contains no duplicates, if `lx` does not contain duplicates, either. If `ly` is infinite, then the resulting list may be partial because the `elem` operation may not terminate.
For finite lists, this implementation is appropriate.

star The output may contain duplicates. If `lx` is infinite, then the generated language is just `T.empty : lx`, so that many elements of `star lx` may not appear in the output.
If `lx` is finite, then `star` can be implemented in a way that guarantees no duplication. However, to retain finiteness, we would have to impose an arbitrary limit on the size of the output.

complement In general there is no computable way to determine whether a word occurs in a lazy list `lx`. Hence, we have no good definition to propose.
If `lx` is finite, then it is possible to enumerate its complement without repetitions. Again, to retain finiteness, we have to impose an arbitrary limit on the size of the output.

Figure 3 contains an implementation of a finite version of the generator module according to the preceding discussion. The implementation of `star` follows the definition of U^* literally. It first recursively creates a list `lxs` of the iterates $U_\bullet^i$ where $U_\bullet = U \setminus \{\varepsilon\}$, concatenates all of them[3], removes the duplicates, and imposes the limit. Removing duplicates introduces quadratic worst-case time complexity in the size of the output.

The implementation of `complement` generates the language Σ^* analogously to the construction of U^* in `star`, uses the list difference operator `L.\\` to remove elements of `lx`, and finally imposes the limit. Its worst-case run time is $O(m \cdot n)$ where m is the limit and n = `length lx`.

In summary, the naive approach in Figure 2 can generate infinite languages, but has many drawbacks that lead to duplication and incompleteness (words in the language are not enumerated). Moreover, the complement is not computable for this approach.

The finite approach in Figure 3 imposes an arbitrary limit on the number of generated words. This limit can lead to omitting words in nonempty languages where $P(w \in R) = 0$. Moreover, there are many places (in `union`, `concatenate`, and `star`) with quadratic worst-case time complexity.

At this point, the question is: Can we do better? Can we come up with a generator that supports finite as well as infinite languages efficiently without incurring extraneous quadratic behavior?

2.3 Ordered Enumeration

First, we concentrate on improving on the quadratic behavior. The key to improve the complexity of union, intersection, and complement lies in representing a language by a strictly increasingly sorted list. In this case, the three operations can be implemented by variations of the merge operation on lists as shown in Figure 4.

[3]It is easy to see that $U^* = U_\bullet^*$.

```haskell
module Examples.Finite where
import qualified Data.Text as T
import Data.List as L

limit :: Int
limit = 1024

union :: Lang -> Lang -> Lang
union lx ly = L.union lx ly

concatenate :: Lang -> Lang -> Lang
concatenate lx ly = L.nub [T.append wx wy | wx <- lx, wy <- ly ]

intersect :: Lang -> Lang -> Lang
intersect lx ly = [wx | wx <- lx, wx `elem` ly ]

star :: Lang -> Lang
star lx = take limit $ removeDuplicates $ concat lxs
  where
    lxs = [T.empty] : map (concatenate lx1) lxs
    lx1 = L.delete T.empty lx
    removeDuplicates [] = []
    removeDuplicates (w:ws) = w : removeDuplicates (filter (/=w) ws)

complement :: Alphabet -> Lang -> Lang
complement sigma lx = take limit (concat lsigmastar L.\\ lx)
  where
    lsigmastar = [T.empty] : map extend lsigmastar
    extend lsigmai = [T.cons a w | a <- sigma, w <- lsigmai]
```

Fig. 3. Finite implementation of the regular operators

The merge-based operations run in linear time on finite lists. However, the operations in Figure 4 are incomplete on infinite lists. As an example of the incompleteness, consider the languages $U = a{\cdot}(a{+}b)^*$ and the singleton language $V = \{b\}$ where $\Sigma = \{a, b\}$ with $a < b$, represented as strictly increasing lists, the infinite list `lu` and the singleton list `lv`. The problem is that the list `union lu lv` does not contain the word b; more precisely, `T.singleton 'b' `elem` union lu lv` does not terminate whereas `u `elem` union lu lv` yields `True` for all `u` in `lu`. The source of the problem is that Haskell's standard ordering of lists and `Text` is the *lexicographic ordering*, which we call $\leq$. It relies on an underlying total ordering on Σ and is defined inductively:

$$\varepsilon \leq v \qquad \frac{u \leq v}{au \leq av} \qquad \frac{a < b}{au \leq bv}$$

This total ordering is often used for Σ^*, but it has the property that there are words $v < w$ such that there are infinitely many words u with $v < u$ and $u < w$. For example, $v = a$, $w = b$, and $u \in U \setminus \{a\}$, which explains the nonterminating behavior just exhibited.

```
union :: (Ord t) => [t] -> [t] -> [t]
union xs@(x:xs') ys@(y:ys') =
  case compare x y of
    EQ -> x : union xs' ys'
    LT -> x : union xs' ys
    GT -> y : union xs ys'
union xs ys = xs ++ ys
```

```
intersect :: (Ord t) => [t] -> [t] -> [t]
intersect xs@(x:xs') ys@(y:ys') =
  case compare x y of
    EQ -> x : intersect xs' ys'
    LT -> intersect xs' ys
    GT -> intersect xs ys'
intersect xs ys = []
```

```
difference :: (Ord t) => [t] -> [t] -> [t]
difference xs@(x:xs') ys@(y:ys') =
  case compare x y of
    EQ -> difference xs' ys'
    LT -> x : difference xs' ys
    GT -> difference xs ys'
difference xs ys = xs
```

Fig. 4. Union, intersection, and difference by merging lists

Fortunately, there is another total ordering on words with better properties. The *length-lexicographic ordering* is defined by $u \leq_{ll} v$ if $|u| < |v|$ or $|u| = |v|$ and $u \leq v$ in the usual lexicographic ordering (but only applied to words of the same length). Here is a definition in Haskell.

```
llocompare :: T.Text -> T.Text -> Ordering
llocompare u v =
  case compare (T.length u) (T.length v) of
    EQ -> compare u v
    LT -> LT
    GT -> GT
```

This ordering has the additional advantage that it gives raise to a standard enumeration of all words over a totally ordered alphabet as an order-preserving bijective function from the natural numbers to Σ^*. Using this bijection we can show that for each pair of words $v \leq_{ll} w$ there is only a finite number of words u such that $v \leq_{ll} u$ and $u \leq_{ll} w$.

For the sake of simplicity, we assume from now on that `T.Text` is ordered by `llocompare` and call the representation of a language by a strictly increasing list in length-lexicographic order its *LLO representation*.

With the LLO representation, the operations `union`, `intersect`, and `difference` run in linear time. If the input languages are finite of size m and n, respectively, then $O(m + n)$ comparisons, pattern matches, and cons operations are needed. Moreover, the operations are complete in the sense that any element in the output is sure to be detected

by a terminating computation. It is easy to implement a version of `elem` that exploits the LLO ordering, such that the element test is decidable for any infinite language.

2.3.1 Concatenation. To implement concatenation, we are in the following situation. Given two languages $U, V \subseteq \Sigma^*$ in LLO representation, produce the LLO representation of $U \cdot V = \{u \cdot v \mid u \in U, v \in V\}$. If we compute the product naively as in Figure 2, then the output is not in LLO form:[4]

```
λ> let lu = ["a", "ab"]
λ> let lv = ["", "b", "bb"]
λ> [ u<>v | u <- lu, v <- lv ]
["a","ab","abb","ab","abb","abbb"]
```

In fact, the output violates both constraints: it is not increasing and it has duplicates.

Perhaps the following observation helps: for each $u \in U$, the LLO representation of the language $u \cdot V$ can be trivially produced because the list `[ u<>v | v <- lv ]` is strictly increasing. This observation motivates the following definition of language concatenation (using `union` from Figure 4).

```
concatenate' :: Lang -> Lang -> Lang
concatenate' lx ly =
  foldr union [] $ [[ T.append x y | y <- ly] | x <- lx]
```

This definition works fine as long as `lx` is finite. If it is infinite, then the `foldr` creates an infinitely deep nest of invocations of `union`, which do not make progress because `union` is strict in both arguments.

At this point, the theory of formal languages can help. The notion of a *formal power series* has been invented to reason about and compute with entire languages [Kuich and Salomaa 1986; Salomaa and Soittola 1978]. In full generality, a formal power series is a mapping from Σ^* into a semiring S and we write $S\langle\langle\Sigma^*\rangle\rangle$ for the set of these mappings. Formally, an element $r \in S\langle\langle\Sigma^*\rangle\rangle$ is written as the formal sum

$$r = \sum_{w \in \Sigma^*} (r, w) \cdot w$$

where $(r, w) \in S$ is the coefficient of w in r. A popular candidate for this semiring is the boolean semiring B because $B\langle\langle\Sigma^*\rangle\rangle$ is isomorphic to the set of languages over Σ. This isomorphism maps $L \subseteq \Sigma^*$ to its characteristic series r_L where $(r_L, w) = (w \in L)$.

The usual language operations have their counterparts on formal power series. We consider just three of them where the "additions" and "multiplications" on the right side of the definitions take place in the underlying semiring.

Sum:	$(r_1 + r_2, w) = (r_1, w) + (r_2, w)$
Hadamard product:	$(r_1 \odot r_2, w) = (r_1, w)(r_2, w)$
Product:	$(r_1 \cdot r_2, w) = \sum_{u \cdot v = w} (r_1, u)(r_2, v)$

Ok, so what is the connection between formal power series and the LLO representation? The LLO representation of a language L can be viewed as a systematic enumeration of the indexes of the non-zero coefficients of the characteristic power series of L. Thus, the `union` operator corresponds to the sum and the `intersect` operator corresponds to the Hadamard product (in B the $+$ and $\cdot$ operators are $\vee$ and $\wedge$).

[4] The example relies on the operator `Data.Monoid.<>` to append strings in any representation.

We also get a hint for computing the product. To find out whether $w \in U \cdot V$ we need to find $u \in U$ and $v \in V$ such that $u \cdot v = w$. Abstracting from this observation, we obtain that building a word w with $|w| = n$ requires two words $u \in U$ and $v \in V$ such that $|u| + |v| = |w| = n$. Hence, it would be useful to have a representation that exposes the lengths of words.

To obtain such a representation, we represent a language as a power series

$$L = \sum_{n=0}^{\infty} L_n x^n$$

where, for all n, $L_n \subseteq \Sigma^n$, a decomposition which corresponds directly to the definition of Σ^*. The language operations can be expressed on this representation similarly as on formal power series.[5]

$$\text{Sum:} \qquad (U + V)_n = U_n \cup V_n \tag{1}$$

$$\text{Hadamard product:} \qquad (U \odot V)_n = U_n \cap V_n \tag{2}$$

$$\text{Product:} \qquad (U \cdot V)_n = \bigcup_{i=0}^{n} U_i V_{n-i} \tag{3}$$

At this point, we arrived at an actionable representation that we call the *segment representation*. By applying the usual spiel of representing a power series by its sequence of coefficients, the definition of product becomes executable.

As a first step, the function `segmentize` transforms the LLO representation into the segment representation. It splits the input into chunks that contain all words of the same length.

```
segmentize :: Lang -> [Lang]
segmentize = collect 0
  where
    collect n lx =
      let (lxn, lxrest) = splitWhile (\xs -> T.length xs == n) lx in
      lxn : collect (n+1) lxrest

splitWhile :: (a -> Bool) -> [a] -> ([a], [a])
splitWhile p [] = ([], [])
splitWhile p xs@(x:xs')
  | p x = let (takes, drops) = splitWhile p xs' in (x:takes, drops)
  | otherwise = ([], xs)
```

As the input sequence to `segmentize` is strictly increasing in the length-lexicographic ordering, the individual segments are strictly increasing, too.

We arrive at the implementation of concatenation in Figure 5. The function `combine` implements the body of the sum in Equation (3). The output of `combine` is strictly increasing because `xsegs !! i` is strictly increasing and all words have the same length `i`. Each of them is prepended to the elements of the strictly increasing sequence `ysegs !! (n - i)`, so that the resulting sequence is strictly increasing.

[5]Not surprisingly, because this representation is based on formal power series taken from $\Sigma^* \langle\langle x^* \rangle\rangle$ considering the semiring of formal languages with union (+) and intersection (·).

```
concatenate :: Lang -> Lang -> Lang
concatenate lx ly = collect 0
  where
    xsegs = segmentize lx
    ysegs = segmentize ly
    collect n =
      (foldr union [] $ map (combine n) [0 .. n]) ++ collect (n+1)
    combine n i =
      [T.append x y | x <- xsegs !! i, y <- ysegs !! (n - i)]
```

Fig. 5. Concatenation for LLO representation

```
star :: Lang -> Lang
star lx = concat rsegs
  where
    xsegs = segmentize lx
    rsegs = [T.empty] : collect 1
    collect n =
      (foldr union [] $ map (combine n) [1 .. n]) : collect (n + 1)
    combine n i =
      [T.append x y | x <- xsegs !! i, y <- rsegs !! (n - i)]
```

Fig. 6. Kleene closure for LLO representation

The function `collect` implements the summation in Equation (3) by taking the union of all `combine n i` for `i` in `[0 .. n]`. As this union has finitely many operands, the nested invocations of `union` do not get stuck.

2.3.2 Kleene Closure. To compute the Kleene closure, the same ideas as for concatenation apply. The star operation is only defined on a *proper* power series where $L_0 = \emptyset$ (so that the language does not contain the empty word ε). In this case $L^* = \bigcup_{n=0}^{\infty} L^n = L^0 + \bigcup_{n=1}^{\infty} L^n = \{\varepsilon\} + L \cdot \bigcup_{n=0}^{\infty} L^n = \{\varepsilon\} + L \cdot L^*$. Surprisingly, this calculation can be turned into an effective algorithm for computing the coefficients of the power series.

$$\text{Star:} \qquad (U^*)_0 = 1 \qquad\qquad (U^*)_n = (U \cdot U^*)_n = \bigcup_{i=1}^{n} U_i \cdot (U^*)_{n-i} \tag{4}$$

The key observation is that Equation (4) is a proper inductive definition of the power series for U^* if we assume that $\varepsilon \notin U$. By this assumption $U_0 = 0$ and the union only touches the coefficients $(U^*)_{n-1}$ down to $(U^*)_0$. Hence, $(U^*)_n$ is well defined as it only relies on U and previously computed indexes!

Figure 6 contains the resulting implementation of Kleene closure. After the discussion of concatenation, there is not much left to say as the `collect` and `combine` functions are almost identical. The key point is to see that this definition is well defined, as we argued.

2.3.3 Complement. The definition of the complement of language U is $\Sigma^* \setminus U$. The `difference` operator in Figure 4 is an implementation of set difference $\setminus$ on LLO represented sets. If `lx` is the LLO representation of U, then `lx` is a suitable argument for `difference`. Looking back at the definition of `lsigmastar` in Figure 3, we see that its output is already in LLO representation. Figure 7 just puts the two definitions together.

```
complement :: Alphabet -> Lang -> Lang
complement sigma lx = difference (concat lsigmastar) lx
  where
    lsigmastar = [T.empty] : map extend lsigmastar
    extend lsigmai = [T.cons a w | a <- sigma, w <- lsigmai]
```

Fig. 7. Complementation for the LLO representation

2.3.4 Discussion. What have we gained? We can generate LLO representations for all regular languages. We can compute concatenation and Kleene closure effectively by transforming the language into a power series and back.

However, there is a catch as the output of `segmentize` is always an infinite list. The advantange of this choice is the simplicity of the code: all index accesses into the lists `xsegs` and `ysegs` are defined. The disadvantage is that `concatenate` returns a partial list if both argument lists are finite. This situation arises in the following example:

```
λ> concatenate [""] [""]
[""
```

With these inputs, only `xsegs !! 0` and `ysegs !! 0` are non-empty, so that for `n>0`

```
collect n = [] ++ collect (n+1)
```

Obviously, such a definition is not productive.

A similar phenomenon arises with `star`, where `star []` and `star [""]` result in partial lists. All other input languages result in infinite languages, which are generated productively.

The next section discusses ways to address these shortcomings.

3 IMPROVEMENTS

3.1 Segmented Representation

Two operations on the LLO representation, `concatenate` and `star`, internally transform their inputs into segment sequences. Such a transformation forth and back seems wasteful, so we stipulate to perform the entire generation process in terms of segment sequences. The transformation to a language only happens as the final step. Thus, the operations in this subsection manipulate languages in terms of the following type:

```
type SegLang = [[T.Text]]
```

As a second refinement, we try to address the productivity concerns discussed in Section 2.3.4. The approach is to let finite segment sequences represent finite languages. Hence, the base cases for the interpretation of regular expression may be defined as in Figure 8. The implementation of `union`, `intersect`, and `difference` (and hence `complement`) is an easy exercise and thus omitted. We discuss the remaining issues with the preliminary code for concatenation in Figure 9. It relies on a special indexing operation into a list of lists that returns an empty list if indexing occurs beyond the end of a segment list:

```
(!!!) :: [[a]] -> Int -> [a]
[]       !!! n = []
(xs:xss) !!! 0 = xs
(xs:xss) !!! n = xss !!! (n - 1)
```

```
zero :: SegLang
zero = []

one :: SegLang
one = [[T.empty]]

atom :: Char -> SegLang
atom t = [[], [T.singleton t]]
```

Fig. 8. Implementation of base cases for segmented language representation

```
concatenate' :: SegLang -> SegLang -> SegLang
concatenate' lx ly = collect 0
  where
    collect n =
      (foldr ILO.union [] $ map (combine n) [0 .. n]) : collect (n+1)
    combine n i =
      [T.append x y | x <- lx !!! i, y <- ly !!! (n - i)]
```

Fig. 9. Preliminary implementation of concatenation for segmented language representation

This change makes `concatenate'` amenable to work with finite segment lists, but any concatenation of languages yields an infinite list of segments. To avoid that we need a termination criterion for `collect` to determine whether any future call can make a non-empty contribution.

To this end, we extend the code to keep track of the lowest index seen in `lx` and `ly` where the respective segment list is exhausted, say, m_x and m_y (if they exist). These indexes are upper bounds for the length of the longest word in `lx` and `ly` such that $\forall x \in$ `lx` it must be that $|x| \leq m_x - 1$ and analogously for `ly`. Then we apply the following lemma.

Lemma 3.1. *Let $X, Y \subseteq \Sigma^*$ be languages and suppose that there are numbers $m_x, m_y \geq 0$ such that*

$$\forall x \in X, |x| < m_x \qquad\qquad \forall y \in Y, |y| < m_y$$

Then $\forall w \in X \cdot Y, |w| < m_x + m_y - 1$.

To prove the lemma, observe that the longest word in the product is bounded by $|xy| = |x| + |y| \leq m_x - 1 + m_y - 1$. In consequence, if $n \geq m_x + m_y - 1$, then no word of length n can be constructed by concatenation of elements from `lx` and `ly`. Figure 10 contains the corresponding extension of `concatenate`. It makes use of `liftA2` used at type `(Int -> Int -> Int) -> Maybe Int -> Maybe Int -> Maybe Int` to add two values of type `Maybe Int` if they are present.[6]

The output of `star` is only finite in two cases as $U^* = \{\varepsilon\}$ iff $U \subseteq \{\varepsilon\}$. Otherwise $|U^*|$ is infinite. The finite cases are straightforward to detect and thus not further discussed.

[6]The function `liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c` is taken from the standard module `Control.Applicative`.

```
concatenate lx ly =
  collect lx ly Nothing Nothing 0
  where
    collect xss yss mmx mmy n =
      let (xss', mmx') = updateMax xss mmx n
          (yss', mmy') = updateMax yss mmy n
          mbound = liftA2 (+) mmx mmy
      in
        case mbound of
          Just m | n >= m - 1 ->
                   []
          _ ->
            (foldr ILO.union [] $ map (combine n) [0 .. n]) :
            collect xss' yss' mmx' mmy' (n+1)
    combine n i =
      [T.append x y | x <- lx !!! i, y <- ly !!! (n - i)]
    updateMax _ mm@(Just _) n = ([], mm)
    updateMax [] Nothing n = ([], Just n)
    updateMax (_:xss) Nothing n = (xss, Nothing)
```

Fig. 10. Finiteness preserving implementation of concatenation

3.2 Assessment

The segmented representation has a number of features that makes it well suited in practice. We discuss these features in turn and consider further minor improvements in the subsections to follow.

3.2.1 Productivity. Each generalized regular expression is interpreted by a productive (finite or infinite) stream of segments. Each segment is guaranteed to be a total, finite list of words. There is no partiality in this representation.

3.2.2 Easy Bounding. To restrict the generated segmented output `segs` to words of length less than a given bound `n`, all we need to do is `concat (take n segs)`. The result is a finite list of words. In contract, such filtering is not effective for the LLO representation where `takeWhile (\w -> T.length w < n) llo` may yield a partial list.

3.2.3 Finite Languages. Segmentation yields a finite representation for all finite languages defined without using the `complement` operation. While $[\![\sim(\sim\mathbf{0})]\!] = \emptyset$ is a finite language, the output of `complement (complement zero)` is an infinite list of empty segments.

In Section 3.5, we discuss ways to get finite representations from more languages, but in general it is not possible to guarantee that a finite language is represented by a finite stream of segments.

3.3 Faster Concatenation

Looking back at the code in Figure 9, we see that the invocation of `collect n` leads to n invocations of the list indexing operations `lx !!! i` and `ly !!! (n - i)`, which results in an overall complexity of $O(n^2)$ for generating the segment n.[7]

There are several different ways to reduce this complexity.

[7] We assume that GHC is sufficiently clever to pull `ly !!! (n - i)` out of the loop.

```
concatenate'' :: SegLang -> SegLang -> SegLang
concatenate'' lx ly = collect ly []
  where
    collect (ysegn:ysegs) rly =
      let rly' = ysegn : rly in
      (foldr ILO.union [] $ zipWith (liftA2 T.append) lx rly') :
      collect ysegs rly'
```

Fig. 11. Concatenation with convolution

3.3.1 Convolution. The `combine` operation is an example of a convolution where indexing of the two lists proceeds in opposite directions. The `collect` operation could take advantage of this pattern and build a reversed list `rly` of already processed segments of `ly` at $O(1)$ extra cost. Using this additional data structure, the convolution can be implemented in linear time using `zipWith` as shown in Figure 11.

For simplicity, the code in the figure only handles infinite lists of segments. But when extended with handling the case for finite lists, it gives rise to a very nice optimization. If `lx` is finite, then the `zipWith` automatically stops processing unnecessary indices in `ly`. Conversely, if `ly` starts with some empty segments because all its words are longer than some lower bound, then these segments could safely be omitted from `rly`.

This consideration reveals that the algorithm implemented in `concatenate''` is inherently asymmetric because it does not exploit the finiteness of `ly`. In the finiteness-aware version, this asymmetry can be addressed easily. The code can be enhanced to detect that `ly` is finite. Then it just flips the roles of `lx` and `ly`, so that from now on `ly` is traversed in forward direction and `lx` backwards. Flipping roles at length n requires taking the initial $(n + 1)$ segments of `lx` and reversing the resulting list. The cost for this reversal is amortized by the previous traversal of `ly` up to this point, so it has no impact on the asymptotic complexity.

We implemented the finiteness-aware algorithm including role flipping, but we do not include the code in the paper, as it is a straightforward, but tedious variation of the code in Figure 11.

3.3.2 Map Data Structure. Alternatively, we could store the segments of `lx` and `ly` up to length `n` in a map data structure of type `Data.Map.Map Int [T.Text]`, which maps the number i to the segment that contains the words of length i. This change would bring down the quadratic complexity to $O(n \log n)$.

3.3.3 Sparse Indexing. Along with the recursive calls of `collect`, we could accumulate a list/set of indexes for `lx` and `ly` such that the corresponding segments are non-empty. With this information, the `combine` operation could be targeted to only consider the useful indexes where both `lx !!! i` and `ly !!! (n - i)` are non-empty.

The combination of using maps and sparse indexing gives rise to a more symmetric behavior.

3.4 Faster Closure

The optimizations described for concatenation also apply to the computation of the Kleene closure. The convolution approach does not require any flipping because it is clear that only the input language can be finite, as the cases where the output language is finite are treated specially, anyway.

Using a map data structure and sparse indexing can be done in exactly the same way as for concatenation.

3.5 More Finite Representations

We already remarked that we can keep segmented representations finite for finite languages constructed without using `complement`. An obvious idea to extend the realm of finite representations is to use a custom data type `Segments` for segment lists.

```
data Segments
  = Empty
  | Cons Lang Segments
  | Full [Lang]
```

The constructor `Empty` represents the empty set. A segment list of the form `Cons xl xsegs` represents the union of the language `xl` and the segments `xsegs`. If such a `Cons` node appears at level $n \geq 0$ in a `Segments` data structure, then all words in `xl` have length n. The constructor `Full xls` is the novelty of this type. If it appears at level n, then it represents all words in Σ^* of length $\geq n$. For convenience, the argument `xls` contains these words, structured as a (standard) list of segments.

The definition of `Segments` relies on a special data type `Lang` to represent languages $L \subseteq \Sigma^n$, for some n.

```
data Lang
  = Null
  | Data [T.Text]
  | Univ [T.Text]
```

The constructor `Null` stands for the empty set, `Data ws` stands for a non-empty set of words represented by an increasingly ordered list `ws`, and `Univ ws`, when encountered at level n, indicates that its argument `ws` represents the full set of word Σ^n.

It is an easy, but tedious exercise to implement the operations `union`, `intersect`, and `difference` on `Lang` and `Segments` in a way that preserves the above invariants as much as possible.

The resulting generation algorithm solves our previous problem with $[\![\sim(\sim\mathbf{0})]\!] = \emptyset$, because it evaluates to `Empty`. Also, $[\![\sim a]\!]$ evaluates to a finite representation:[8]

```
Cons (Univ [""]) (Cons (Data ["b"]) (Full _))
```

But a slight variation like $[\![(\sim a)(\sim b)]\!] = \{a, b\}^*$ would not be represented finitely.

It turns out that we can extend the range of languages for which finite representations are generated by dualizing the idea for detecting finiteness in Section 3.1. The following lemma captures this idea.

Lemma 3.2. *Let $X, Y \subseteq \Sigma^*$ be languages and suppose that there are numbers $f_x, f_y \geq 0$ such that*

$$\forall x \in \Sigma^*, |x| \geq f_x \Rightarrow x \in X \qquad\qquad \forall y \in \Sigma^*, |y| \geq f_y \Rightarrow y \in Y$$

Then $\forall w \in \Sigma^*, |w| \geq f_x + f_y \Rightarrow w \in X \cdot Y$.

An algorithm analogous to the finiteness-preserving one in Figure 10 can determine f_x and f_y by detecting when a segment list is `Full`. Once f_x and f_y are both determined and the generated word length n is greater than or equal to $f_x + f_y$, then we can finish the segments by outputting the appropriate `Full`.

[8] We use string notation for elements of `Data.Text.Text` for readability. The bar in `Full _` stands for the infinite list of full segments that need not be evaluated.

This algorithm generates finite segmented representations for many finite and co-finite languages. For example, the computed representations for $\sim a$, $\sim b$, and $(\sim a)(\sim b)$ are as follows.

```
Cons (Univ [""]) (Cons (Data ["b"]) (Full _))
Cons (Univ [""]) (Cons (Data ["a"]) (Full _))
Cons (Univ [""]) (Cons (Data ["a", "b"]) (Full _))
```

But the algorithm is easy to defeat. For example, consider $[\![a^* + \sim(a^*)]\!] = \Sigma^*$ or $[\![a^* \,\&\, \sim(a^*)]\!] = \emptyset$, which are both mapped to infinite segment lists.

4 OCAML IMPLEMENTATION

In addition to the Haskell implementation, we implemented the language generator in OCaml. The OCaml version only implements the "latest" version of the algorithm with a segmented representation and fast backward lookup and convolutions for concatenation and star. It however doesn't implement the symbolic representation of segments discussed in Section 3.5. The main goal of this implementation is to experiment with strictness and various data structures for segments. The key idea is that the internal order on words in a segment does not matter, because each segment only contains words of the same length. All we need is a data structure that supports the power series operations.

To facilitate such experimentation, we implemented the generator algorithm as a functor whose signature is shown in Figure 12. The functor takes two data structures as arguments: words and segments. This structuring allows us to test numerous representations easily without changing the code. It also forced us to find the "minimal" set of operations needed to implement the algorithm.

Characters and Words. Figure 12b contains the signature for words. Surprisingly few operations are needed. We need to build the empty word (for `One`), singleton words (for `Atom`), and to append two words. Neither an ordering nor a length operation is needed for words: Comparison is encapsulated in the segment data structure and the length of a word is the index of the segment in which it appears.

This signature is easily satisfied by the OCaml `string` type (i.e., arrays of bytes), arrays, lists of characters, or ropes. The type of individual characters is unrestricted.

Segments. Figure 12a contains the signature for segments. The first group of operations creates and tests for empty segments and singleton segments. The main requirement is to support the operations on power series described in subsection 2.3. For the set operations, we need the functions `union`, `inter` and `inter`. The product described in Equation 3 is decomposed in two parts:

- An `append` function which implements $U_i V_{n-i}$. It computes the product of two segments by pairwise appending their elements.
- A `merge` operation which computes the union of an arbitrary number of segments. Its purpose is to collect the segments obtained by invocations of `append`.

As the goal is to experiment with different data structures, we want to use implementations that are transient by default. We thus require a function `memoize` that avoids recomputing segments accessed multiple times, which happens when computing the concatenation or the closure of languages. Finally, the functions `of_list` and `iter` import and export elements to and from a segment.

```
module type SEGMENT = sig
  type elt (** Elements *)
  type t (** Segments *)

  val empty : t
  val is_empty : t -> bool
  val return : elt -> t

  (** Set operations *)
  val union : t -> t -> t
  val inter : t -> t -> t
  val diff : t -> t -> t

  (** Product *)
  val append: t -> t -> t

  (** n-way merge *)
  val merge : t list -> t

  (** Import a list *)
  val of_list : elt list -> t

  (** Export elements *)
  val iter : t -> (elt -> unit) -> unit

  (** For transient data-structures *)
  val memoize : t -> t
end
```

(a) Operations on segments

```
module type WORD = sig
  type char
  type t
  val empty : t
  val singleton : char -> t
  val append : t -> t -> t
end
```

(b) Operations on words

```
module Regenerate
    (Word : WORD)
    (Segment : Segments.S with type elt = Word.t)
: sig
  type lang = Segment.t stream
  val gen :
    sigma:Segment.t -> C.t regex -> lang
  val iter : lang -> (Word.t -> unit) -> unit
end
```

(c) Language generation as a functor

Fig. 12. Signatures of the language generator

4.1 Core Algorithm

The core algorithm is very similar to the Haskell version. The power series is implemented using a thunk list in the style of Pottier [2017]:

```
type 'a node =
  | Nil
  | Cons of 'a * 'a stream
type 'a stream = unit -> 'a node
```

A stream is represented by a function which takes a unit argument and returns a node. A node, in turn, is either `Nil` or a `Cons` of an element and the tail of the stream. The empty stream, for instance, is represented as `fun () -> Nil`. This representation has several advantages[9]: it is lazy, fast, lightweight, and almost as easy to manipulate as regular lists. It gives rise to a very natural representation for languages represented as power series:

```
type lang = Segment.t stream
```

The rest of the implementation is similar to the Haskell one. For instance, here is the implementation of the union of languages:

```
let rec union s1 s2 () = match s1(), s2() with
  | Nil, x | x, Nil -> x
  | Cons (x1, next1), Cons (x2, next2) ->
```

[9] See https://github.com/ocaml/ocaml/pull/1002 for a long discussion on the topic.

```
    Cons (Segment.union x1 x2, union next1 next2)
```

The trailing unit argument, `()`, is essential because it allows us to drive the evaluation of the stream lazily. With this definition, `union s1 s2` will not cause any evaluation until it is applied to `()`.

The concatenation of languages demonstrates how the main algorithm can be expressed once the segment operations have been abstracted:. To build $U \cdot V$, we first build the nth term $(U \cdot V)_n = \bigcup_{i=0}^{n} U_i V_{n-i}$. We use both `Segment.append` to implement the product of segments and the concatenation of words and `Segment.merge` to merge all the resulting segments.

```
let term_of_length map1 map2 n =
  let combine_segments i =
    Segment.append (IntMap.find i map1) (IntMap.find (n - i) map2)
  in
  List.(range 0 n) |> List.rev_map combine_segments |> Segment.merge
```

We then collect all the terms by synchronized recursion over the power series U and V:

```
let rec collect n map1 map2 seq1 seq2 () = match seq1 (), seq2 () with
  | Cons (segm1, seq1), Cons (segm2, seq2) ->
    let map1 = IntMap.add n (Segment.memoize segm1) map1 in
    let map2 = IntMap.add n (Segment.memoize segm2) map2 in
    Cons (term_of_length map1 map2 n, collect (n+1) map1 map2 seq1 seq2)
```

The `IntMap` module provides a functional implementation of maps keyed by integers. The maps are used to quickly access segments of smaller index that have been computed in earlier invocations of `collect`. As such segments are accessed multiple times, we use `memoize` to avoid computing them over and over again. Functional maps are sufficient because the size of a map is equal to the maximum word length, which does not get excessively large.

Finally, we initialize `collect` with empty maps.

```
let concatenate = collect 0 IntMap.empty IntMap.empty
```

The biggest difference compared to the Haskell version relates to the use of recursion to define `star`-related operations. For instance, the Haskell implementation defines `lsigmastar` as `[T.empty] : map extend lsigmastar`. Such a recursive definition can not be translated directly to OCaml, since it is not compatible with strict-by-default evaluation and will be prevented by the compiler. Instead, we simply use an accumulator to keep track of the latest version of the language. A similar change was made for the definition of `star`.

```
let sigma_star sigma =
  let rec collect acc () =
    Iter.Cons (acc, collect (Segment.append sigma acc))
  in
  collect (Segment.return W.empty)
```

4.2 Data Structures

Now that we implemented the language generator parameterized by segments, we can experiment with various data structures for segments. We present a range of potential implementations before comparing their performance.

4.2.1 Ordered Streams. The Haskell implementation in subsection 2.3 represents segments as ordered enumerations. We can use the same representation in OCaml thanks to the `stream` datatype.

```
type t = elt stream
```

To use an order, we need a comparison and an append function on words. The `OrderedMonoid` signature captures these requirements. Figure 13 shows the resulting functor `ThunkList`.

```
module type OrderedMonoid = sig
  type t
  val compare : t -> t -> int
  val append : t -> t -> t
end
module ThunkList (Elt : OrderedMonoid) : SEGMENTS with type elt = Elt.t
```

Fig. 13. Signature for `ThunkList`

Most of the implementation is straightforward. For example, here is the append function, where `>>=` is bind (or `concatMap`) and `>|=` is map.

```
let append l1 l2 =
  l1 >>= fun x -> l2 >|= fun y -> Elt.append x y
```

The n-way merge is implemented using a priority heap which holds pairs composed of the head of a stream and its tail. When a new element is required in the merged stream, we pop the top element of the heap, deconstruct the tail and insert it back in the heap.

```
let merge l =
  let cmp (v1,_) (v2,_) = K.compare v1 v2 in
  let merge (x1, s1) (_, s2) = (x1, s1@s2) in
  let push h s =
    match s() with Nil -> h | Cons (x, s') -> Heap.insert h (x, [s'])
  in
  let h0 = List.fold_left push (Heap.empty ~cmp ~merge) l in
  let rec next heap () =
    if Heap.is_empty heap then Nil else begin
      let (x, seq), heaps = Heap.pop heap in
      let new_heap = List.fold_left push heaps seq in
      Cons (x, next new_heap)
    end
  in
  next h0
```

4.2.2 Transience and Memoization. During concatenation and star, we iterate over segments multiple times. As thunk lists are transient, iterating multiple times over the same list will compute it multiple times. To avoid this recomputation, we can implement memoization over thunk lists by pushing the elements in a growing vector as they are computed.

Before evaluating a new thunk, we first check if it is already available in the vector. Otherwise, evaluate it, push it into the vector and return it.

```
let memoize f =
  let r = CCVector.create () in
  let rec f' i seq () =
    if i < CCVector.length r
    then CCVector.get r i
    else
      let l = match seq() with
        | Nil -> Nil
        | Cons (x, tail) -> Cons (x, f' (i+1) tail)
      in
      CCVector.push r l;
      l
  in
  f' 0 f
```

Such a memoization function incurs a linear cost on streams. To test if this operation is worthwhile we implemented two modules: `ThunkList` where memoization is the identity and `ThunkListMemo` with the implementation above.

4.2.3 Lazy Lists. OCaml also supports regular lazy lists using the builtin `Lazy.t` type:

```
type 'a node =
  | Nil
  | Cons of 'a * 'a lazylist
type 'a lazylist = 'a node Lazy.t
```

We implemented a `LazyList` functor which is identical to the `ThunkList` but uses lazy lists.

4.2.4 Strict Sets. As the main operations on segments are set operations, one might expect a set implementation to perform well. We implemented segments as sets of words using OCaml's built-in `Set` module. OCaml sets are implemented using binary trees with an imbalance of at most one. The only operations not implemented by OCaml's standard library are the n-way merge and the product, which can be implemented using folds and unions.

4.2.5 Tries. Tries [Fredkin 1960] are prefix trees where each branch is labeled with a character and each node may contain a value. A trie represents a mapping from a set of words and a word belongs to its domain if there is a path reaching a value labeled with the characters in the word. Tries seem well adapted to our problem:

- All words in a segment have the same length. As no prefixes need to be represented, we can elide the values at inner nodes and tries where values are only at the leaves.
- The `append` operation on tries can be implemented by grafting the second trie to all the leaves of the first one.

Hence, we can implement tries similarly to tries of integers as described by Okasaki and Gill [1998]. For simplicity, we do not use path compression, which means that branches are always labeled with one character. A trie is then either `Empty`, a `Leaf` containing a value, or a `Node` containing a map from characters to its child tries. As we are only interested in the paths, we use tries with unit values.

```
type 'a trie =
  | Empty
  | Leaf of 'a
  | Node of 'a trie CharMap.t

type t = unit trie
```

The implementation of most operations follows the literature closely. The only novel operation is append to compute the product of two sets. It can be implemented in a single traversal.

```
let rec append t t0 = match t with
  | Empty -> Empty
  | Leaf () -> t0
  | Node map -> M.map (fun t' -> append t' t0) map in
```

The implementation of append relies on the fact that we do not store values at the leaves. Hence, the appended trie `t0` need not by copied, which is more efficient both in terms of time and space as it leads to a high degree of sharing between the tries.

5 BENCHMARKS

This section compares the performance of our implementations in two dimensions. First, we compare the successive algorithmic refinements using the Haskell implementation presented in Section 2 and Section 3. Second, we compare different choices of data structures to represent segments using the OCaml implementation described in Section 4.

All benchmarks were done on a ThinkPad T470 with an i5-7200U CPU and 12G of memory. The Haskell benchmarks use the Stackage LTS 10.8 release and the `-O2` option. The OCaml benchmarks use OCaml 4.06.1 with the flambda optimizer and the `-O3` option. We annotate functors with the `[@inline]` annotation to ensure that functors are applied at compile time and their content benefits from available optimizations.

5.1 Comparing Algorithms in the Haskell Implementation

Section 2 and Section 3 develop the algorithm for generating languages in a sequence of changes applied to a naive baseline algorithm. We now evaluate the impact of these changes on performance, which we plan to measure in terms of generation speed in words per second. It turns out that this speed depends heavily on the characteristics of the the regular expression considered. For that reason, we choose three representative regular expressions to highlight the strengths and weaknesses of the different approaches.

- a^*: This expression describes a very small language with $P(w \in L) = 0$. Nevertheless, it puts a lot of stress on the underlying append operation on words as their length increases very quickly. The input language contains only one segment whereas all segments of the output language contain exactly one element. This combination highlights the usefulness of sparse indexing and maps.
- $(a \cdot b^*)^*$: On the opposite end of the spectrum, the language of this regular expression is fairly large with $P(w \in L) = 0.5$. The expression applies `star` to a language where segment $n + 1$ consists of the word ab^n. Its evaluation measures the performance of `star` on a non-sparse language and of concatenation applied to a finite and an infinite language.

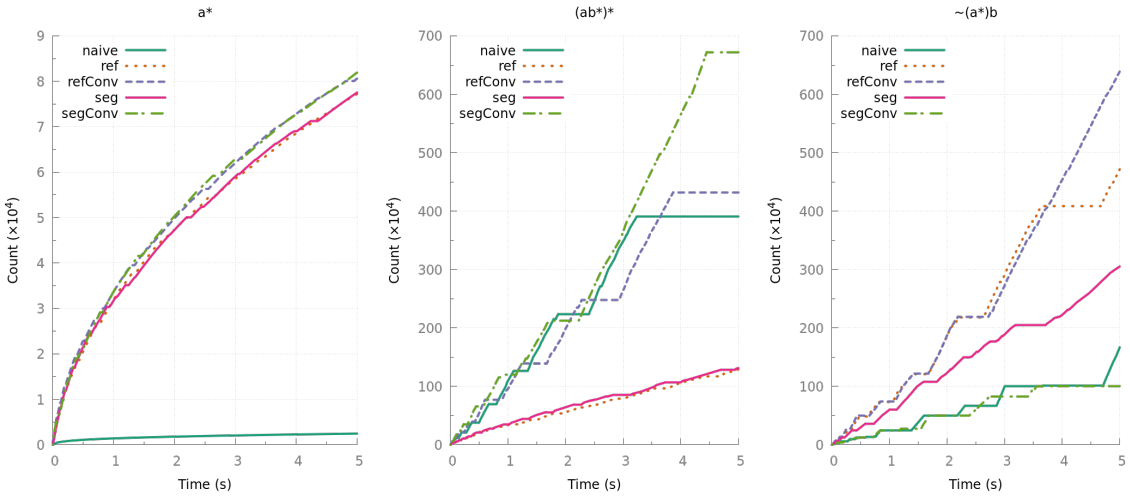


Fig. 14. Benchmark for the Haskell implementation with various algorithms

- $\sim(a^*) \cdot b$: Finally, this regular expression exercises the complement operation and tests the concatenation of a very large language, $P(w \in [\![\sim(a^*)]\!]) = 1$, to a much smaller language.

In the evaluation, we consider five variants of the Haskell implementation.

- The **naive** implementation corresponds to the code developed by the end of Section 2. It transforms to and from segments on the fly and it uses plain list indexing in the implementation of concatenation and closure.
- The **seg** implementation uses the infinite list-based segmented representation throughout (Section 3.1). Moreover, it relies on using maps and sparse indexing (Sections 3.3.2 and 3.3.3) for concatenation and closure.
- The **segConv** implementation uses finite and infinite segment lists, but it applies the convolution approach (Section 3.3.1) to implement concatenation and closure.
- The **ref** implementation uses the specialized segmented representation from Section 3.5 combined with maps and sparse indexing.
- The **refConv** implementation uses the specialized segmented representation from Section 3.5 combined with convolution.

To evaluate the performance, we iterate through the stream of words produced by the generator and force their evaluation.[10] Every 20 words, we record the time elapsed since the start of the iteration. We stop the iteration after 5 seconds. The resulting graph plots the time (x-axis) against the number of words (y-axis) produced so far. The slope of the graph in indicates the generation speed of the plotted algorithm, high slope is correlated to high generation speed. Figure 14 contains the results for the Haskell implementations.

Most algorithms generate between 3000 and 150000 words in the first second, which seems more than sufficient for testing purposes. Regarding comparative performance, the **refConv** implementation which uses symbolic segments and convolutions is consistently in the leading group. Sometimes one of the other implementations is slightly faster, but not consistently so. This observation validates that the changes proposed in Section 3 actually lead to improvements and suggests that **refConv** is the overall winner of this comparison.

[10] In Haskell, forcing is done using `Control.DeepSeq`.

Looking at each graph in more detail, we can make the following remarks:

- All implementations are equally fast on a^* except the naive implementation, which relies on list lookups without sparse indexing.
- For $(a \cdot b^*)^*$ and $\sim(a^*) \cdot b$, the graph of some implementations has the shape of "skewed stairs". We believe this phenomenon is due to insufficient laziness: when arriving at a new segment, part of the work is done eagerly which causes a plateau. When that part is done, the enumeration proceeds lazily. As laziness and GHC optimizations are hard to control, we did not attempt to correct this.
- $(a \cdot b^*)^*$ demonstrates that sparse indexing does degrade performance when applying `star` to non-sparse languages. Using the convolution technique presented in Section 3.3.1 resolves this problem.
- The **ref** and **refConv** algorithms are significantly faster on $\sim(a^*) \cdot b$ compared to **seg** and **segConv**. We have no good explanation for this behavior as the code is identical up to the symbolic representation of full and empty segments. However, the only sublanguage where this representation could make a difference is $[\![b]\!]$, which is also represented finitely by **segConv** and should thus benefit from the convolution improvement in the same way as **refConv**.

5.2 Comparing Data Structures in the OCaml Implementation

By now, we have established that the **refConv** algorithm provides the best overall performance. The Haskell implementation, however, essentially uses lazy lists to represent single language segments. To measure the influence of strictness and the choice of data structure on language generation, we consider the functorized OCaml implementation, as it facilitates such experimentation. We follow the same methodology as the Haskell evaluation using the regular expressions a^*, $(a \cdot b^*)^*$ and $\sim(a^*) \cdot b$. The results are shown in Figure 15.

Unlike the previous benchmark for algorithms, there is no clear winner among the data structures. The `ThunkList` and `LazyList` modules seem to be superior to the alternatives, although the results are highly influenced by the regular expression considered.

- The `Trie` module is very inefficient on a^* because our implementation of tries does not use path compression. In the case of a^*, where each segment contains only one word, the trie degenerates to a list of characters. We believe an implementation of tries with path compressions would perform significantly better.
- The other data structures exhibit a very pronounced slowdown on a^* when reaching 150000 words. We believe this slowdown is due to garbage collection because the active heap contained 10G of data before a collection was triggered. Much less memory is consumed for the other regular expression.
- The strict data structures showcase a very marked "skewed stair" pattern, which is completely absent for `ThunkList` and `LazyList`. Thus, manual control of laziness works very well in OCaml. These results also demonstrate that strict data structures should only be used when all elements up to a given length are needed. In such a case the stair pattern causes no problems.
- Memoization for thunk lists significantly degrades performance. It seems that the linear cost of memoizing the thunk list and allocating the vectors is higher than simply recomputing lists.

While the regular expressions presented previously do exercise `concatenation` and `star`, they do not exercise set operations. To test set operations on non-trivial segments (that are neither full nor empty), we consider the language of words with at least one a and one b. This language can be built in two ways: $\sim(a^*)$ & $\sim(b^*)$ and $(aa^*b + bb^*a) \cdot \Sigma^*$. The first expression applies intersection to two large languages, the second expression takes the union of smaller languages, but uses a concatenation. The performance of the various data structures on these two regular expressions are presented

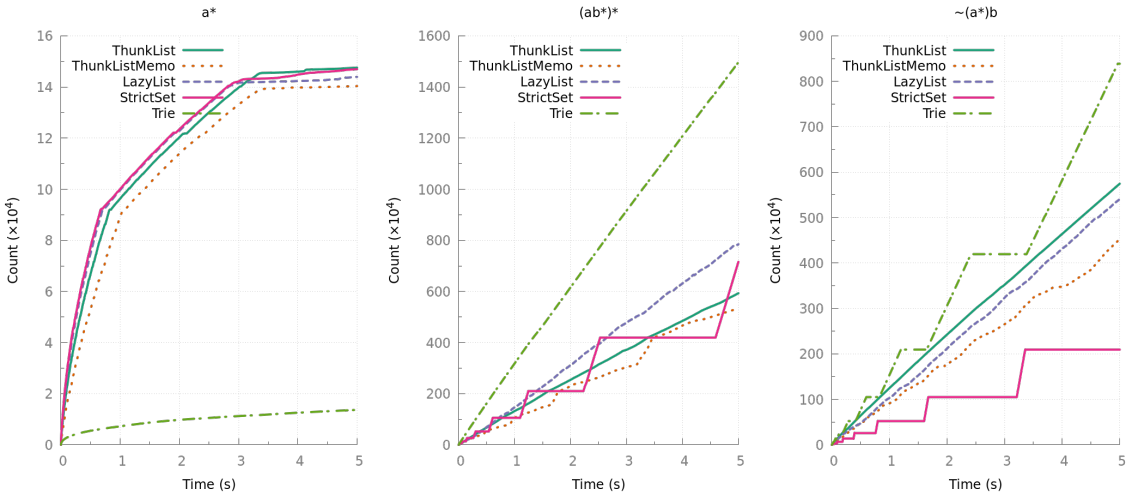


Fig. 15. Benchmark for the OCaml implementation with various data-structures

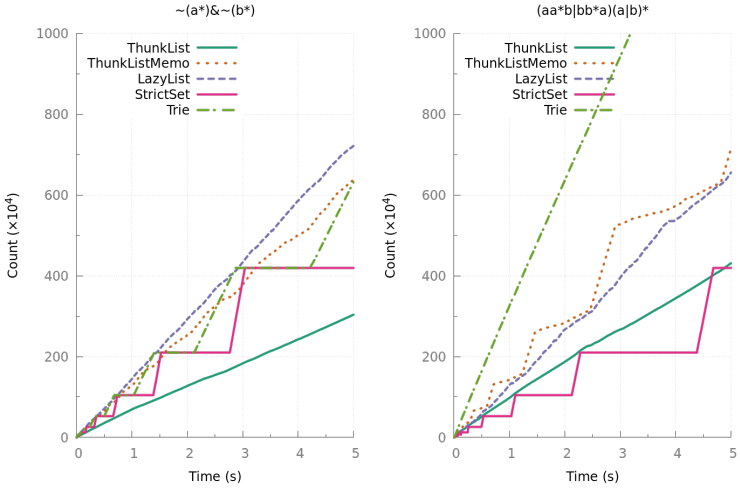
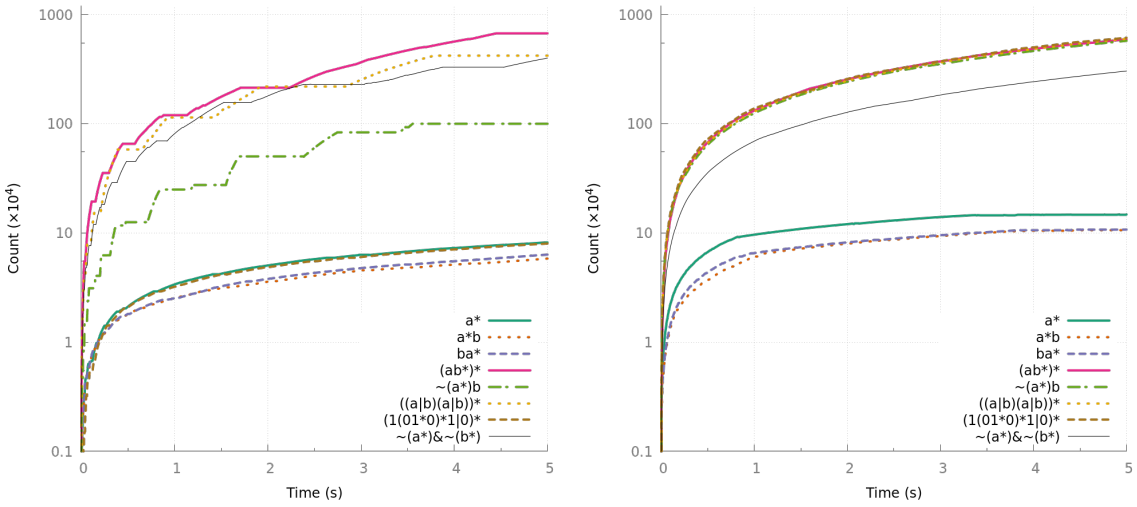


Fig. 16. Benchmarking `union` in the OCaml data-structures

in Figure 16. Lazy and thunk lists, with or without memoization, are very efficient on the union and intersection of languages but less so when concatenation is involved. Performance of strict sets and tries is surprisingly poor.

5.3 The Influence of Regular Expressions on Performance

The benchmark results so far demonstrate that the performance of the language generator highly depends on the structure of both the generated language and the regular expression considered. To further explore this observation we compare a range of regular expressions with the `refConv` Haskell implementation and the `ThunkList` OCaml implementation. Before presenting the results, a word of warning: We do not claim to offer a fair comparison between



(a) Haskell implementation with `segConv`

(b) OCaml implementation with `ThunkList`

Fig. 17. Benchmark on different regular expressions

languages! The two implementations are not exactly the same and we made no attempt to measure both language under exactly the same conditions.

Figure 17 contains the resulting graphs. These graphs use a logarithmic scale for the word count as it enables better comparison between the regular expression specimens.

We add three new regular expressions to the range of expressions already considered:

- $(\Sigma\Sigma)^*$, the language of words of even length. This language is neither finite nor cofinite, but it can make good use of the symbolic representation of segments.[11]
- $(1(01^*0)^*1 + 0)^*$, the language of multiples of 3 written in binary representation. Again, this is a language that is neither finite nor cofinite, but its segments are never full nor empty.
- a^*b and ba^*, which together check whether concatenation behaves symmetric with respect to performance.

Languages are roughly ordered by size/density, i.e., $P(w \in L)$. We observed that the bigger the segments of a language, the faster it is to generate its words. If each segment contains many words, we do not need to compute many segments to generate a large number of words. Moreover, most operations, notably those involving the product of segments, are more expensive when considering segments of higher indices. Briefly put, long strings are harder to generate than short ones.

Regarding symmetry, we find that the generation of a^*b and ba^* has the same performance in the Haskell implementation. The OCaml version does not behave symmetrically, because it does not implement the improved convolution technique with detection of finite languages, as described in Section 3.3.1.

[11] This language has $P(w \in L) = 0.5$ thus solving the exercise posed earlier.

6 RELATED WORK

Regular Language Generation. McIlroy [2004] implements the enumeration of strings of regular languages in Haskell. He develops two approaches, one based on interpreting regular expressions, the other using an intermediate representation as a nondeterministic finite automaton. The approach is inspired by an earlier note by Misra [Misra 2000] and uses operators based on an increasing list representation similar to our first proposal.

However, the approach does not consider the complement operation and none of the presented representations can readily deal with it. Moreover, the generation method is reported to be very inefficient and thus not suitable for generating test inputs at a large scale.

Lee and Shallit [2004] discuss enumerating regular expressions and their languages. Despite the title, this work is unrelated because it aims to find bounds on the number of languages that can be represented with regular expressions and automata of a certain size.

Language Generation. Some authors discuss the generation of test sentences from grammars for exercising compilers (e.g., [Paracha and Franek 2008; Zheng and Wu 2009] for some recent work). This line of work goes back to Purdom's sentence generator for testing parsers [Purdom 1972], which creates sentences from a context-free grammar using each production at least once.

Compared to our generator, the previous work starts from context-free languages and aims at testing the apparatus behind the parser, rather than the parser itself. Hence, it focuses on generating positive examples, whereas we are also interested in counterexamples.

Grammar Testing [Lämmel 2001] aims to identify and correct errors in a grammar by exercising it on example sentences. The purpose is to recover "lost" grammars of programming languages effectively. Other work [Li et al. 2004] also targets testing the grammar, rather than the parser.

Test Data Generation. Since the introduction of QuickCheck [Claessen and Hughes 2000], property testing and test-data generation has been used successfully in a wide variety of contexts. In property testing, input data for the function to test is described via a set of combinators while the actual generation is driven by a pseudo-random number generator. One difficulty of this approach is to find the correct distribution of inputs that will generate challenging test cases. This problem already arises with recursive data types, but it is even more pronounced when generating test inputs for regular expressions because, as explained in Section 2, many languages have a density of zero, which means that a randomly generated word almost never belongs to the langer. Generating random *regular expressions* is much easier. We can thus combine property testing to generate regular expressions and then apply our language generator to generate targeted positive and negative input for these randomly generated regular expressions.

New et al. [2017] are concerned with the enumeration of elements of various data structures. Their approach is complementary to test-data generators. They exploit bijections between natural numbers and the data domain and develop a quality criterion for data generators based on a notion of fairness. It would be interesting to investigate the connection between their enumeration strategies and a direct representation of formal power series.

Crowbar [Dolan and Preston 2017] is a library that combines property testing with fuzzing. In QuickCheck, the generation is driven by a random number generator. Crowbar replaces this generator by `afl-fuzz` [Zalewski 2014]. Afl is a fuzzer that relies on runtime instrumentation to provide good code coverage, thus eliminating the need to specify the distribution of random generators. This approach, however, is not sufficient to generate both regular expressions

and inputs, as we would still require an oracle. Our language generator could allow to easily fuzz regular expression parsers.

7 CONCLUSIONS AND FUTURE WORK

In this article, we presented an algorithm to generate the language of a *generalized* regular expression with union, intersection and complement operators. Using this technique, we can generate both positive and negative instance of a regular expression, thus enabling easier testing of regular expression parsers without an oracle. We provide two implementations: one in Haskell which explores different algorithmic improvements and one in OCaml which evaluates choices in data structures. We then measured the performance of these implementations.

Even though our implementations are not heavily optimized, our approach generates languages at a rate that is more than sufficient for testing purposes, between $1.3 \cdot 10^3$ and $1.4 \cdot 10^6$ strings per seconds. We can now to combine our generator with property based testing to test regular expression parsers on randomly-generated regular expressions. While our approach eliminated the need for an oracle, the burden of correctness now lies on the language generator. We would like to implement our algorithm in Agda and prove its correctness and its productivity.

We also want to extend our approach to a more general context. Notably, we can consider new regular expression operators. While some operators are very easy to implement, such as option and generalized repetition, other would cause some additional challenge, such as lookaround and boundaries operators. We can also consider non-regular operators! Indeed, our approach is compositional, and can thus scale to decidable languages. It remains to be seen if context free languages can also be generated with this method. Representing regular languages as formal power series provides numerous avenues for extensions. In this article, we used the semiring of languages of size n. Other semirings might also yield interesting algorithms. For instance, using the stream of boolean coefficients would allow to generate the language and its complement in one go. We could also use a richer semiring to generate *weighted* words.

Finally, in Section 5, we saw that the performance of our generator highly depends both on the shape of the language and the shape of the regular expression. We noticed a rough correlation with the size of each segments, but did not provide a more precise account. One might wonder if such an account is possible and if yes, can we use this information to improve language generation by transforming a regular expression to an equivalent yet more efficient one.

REFERENCES

Valentin M. Antimirov. 1996. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science* 155, 2 (1996), 291–319.

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494.

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), September 18-21, 2000.*, Martin Odersky and Philip Wadler (Eds.). ACM, Montreal, Canada, 268–279. https://doi.org/10.1145/351240.351266

Russ Cox. 2007. Implementing Regular Expressions. (2007). https://swtch.com/~rsc/regexp/.

Russ Cox. 2010. Regular Expression Matching in the Wild. (March 2010). https://swtch.com/~rsc/regexp/regexp3.html.

Stephen Dolan and Mindy Preston. 2017. Testing with Crowbar. (2017).

Sebastian Fischer, Frank Huch, and Thomas Wilke. 2010. A play on regular expressions: functional pearl. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 357–368. https://doi.org/10.1145/1863543.1863594

Edward Fredkin. 1960. Trie Memory. 3 (Sept. 1960), 490–499. Issue 9. https://doi.org/10.1145/367390.367400

Benoît Groz and Sebastian Maneth. 2017. Efficient testing and matching of deterministic regular expressions. *J. Comput. Syst. Sci.* 89 (2017), 372–399. https://doi.org/10.1016/j.jcss.2017.05.013

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2003. *Introduction to automata theory, languages, and computation - international edition (2. ed).* Addison-Wesley.

Werner Kuich and Arto Salomaa. 1986. *Semirings, Automata, Languages.* EATCS Monographs on Theoretical Computer Science, Vol. 5. Springer. https://doi.org/10.1007/978-3-642-69959-7

Ralf Lämmel. 2001. Grammar Testing. In *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science)*, Heinrich Hußmann (Ed.), Vol. 2029. Springer, 201–216. https://doi.org/10.1007/3-540-45314-8_15

Jonathan Lee and Jeffrey Shallit. 2004. Enumerating Regular Expressions and Their Languages. In *Implementation and Application of Automata, 9th International Conference, CIAA 2004, Kingston, Canada, July 22-24, 2004, Revised Selected Papers (Lecture Notes in Computer Science)*, Michael Domaratzki, Alexander Okhotin, Kai Salomaa, and Sheng Yu (Eds.), Vol. 3317. Springer, 2–22. https://doi.org/10.1007/978-3-540-30500-2_2

Hu Li, Maozhong Jin, Chao Liu, and Zhongyi Gao. 2004. Test Criteria for Context-Free Grammars. In *28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, 27-30 September 2004, Hong Kong, China, Proceedings.* IEEE Computer Society, 300–305. https://doi.org/10.1109/CMPSAC.2004.1342847

M. Douglas McIlroy. 2004. Enumerating the strings of regular languages. *J. Funct. Program.* 14, 5 (2004), 503–518. https://doi.org/10.1017/S0956796803004982

Jayadev Misra. 2000. Enumerating the Strings of a Regular Expression. (Aug. 2000). https://www.cs.utexas.edu/users/misra/Notes.dir/RegExp.pdf.

Max S. New, Burke Fetscher, Robert Bruce Findler, and Jay A. McCarthy. 2017. Fair enumeration combinators. *J. Funct. Program.* 27 (2017), e19. https://doi.org/10.1017/S0956796817000107

Chris Okasaki and Andrew Gill. 1998. Fast Mergeable Integer Maps. In *In Workshop on ML.* 77–86.

A. M. Paracha and Frantisek Franek. 2008. Testing Grammars For Top-Down Parsers. In *Innovations and Advances in Computer Sciences and Engineering, Volume I of the proceedings of the 2008 International Conference on Systems, Computing Sciences and Software Engineering (SCSS), part of the International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering, CISSE 2008, Bridgeport, Connecticut, USA*, Tarek M. Sobh (Ed.). Springer, 451–456. https://doi.org/10.1007/978-90-481-3658-2_79

François Pottier. 2017. Verifying a Hash Table and its Iterators in Higher-Order Separation Logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 3–16. https://doi.org/10.1145/3018610.3018624

Paul Purdom. 1972. A Sentence Generator for Testing Parsers. *BIT* 12, 3 (1972), 366–375. https://doi.org/10.1007/BF01932308

RegExLib [n. d.]. Regular Expression Library. http://www.regexlib.com/. ([n. d.]).

Arto Salomaa and Matti Soittola. 1978. *Automata-Theoretic Aspects of Formal Power Series.* Springer. https://doi.org/10.1007/978-1-4612-6264-0

Ken Thompson. 1968. Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422. https://doi.org/10.1145/363347.363387

M. Zalewski. 2014. (2014). http://lcamtuf.coredump.cx/afl/

Lixiao Zheng and Duanyi Wu. 2009. A Sentence Generation Algorithm for Testing Grammars. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2009, Seattle, Washington, USA, July 20-24, 2009. Volume 1*, Sheikh Iqbal Ahamed, Elisa Bertino, Carl K. Chang, Vladimir Getov, Lin Liu, Hua Ming, and Rajesh Subramanyan (Eds.). IEEE Computer Society, 130–135. https://doi.org/10.1109/COMPSAC.2009.193