Regenerate: A Language Generator for Extended Regular Expressions with an application to test case generation

Gabriel Radanne, Peter Thiemann

Regenerate: A Language Generator for Extended Regular Expressions

with an application to test case generation

Gabriel Radanne
University of Freiburg
Germany
radanne@informatik.uni-freiburg.de

Peter Thiemann
University of Freiburg
Germany
thiemann@acm.org

Abstract

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 belongs to the language described by a regular expression are well explored, yet new algorithms appear frequently. However, there is no satisfactory methodology for testing such matchers.

We propose a testing methodology which is based on generating positive as well as negative examples of words in the language. To this end, we present a new algorithm to generate the language described 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 testing.

CCS Concepts • **Theory of computation → Regular languages**; *Grammars and context-free languages*; • **Software and its engineering → Software testing and debugging**; *Functional languages*;

Keywords Regular expressions; parsing; formal languages; power series; Haskell; OCaml

ACM Reference Format:
Gabriel Radanne and Peter Thiemann. 2018. Regenerate: A Language Generator for Extended Regular Expressions: with an application to test case generation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3278122.3278133

GPCE '18, November 5–6, 2018, Boston, MA, USA

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 derivatives [Brzozowski 1964] and map a regular expression directly to a deterministic automaton. Partial derivatives [Antimirov 1996] provide 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. Static test suites are somewhat unsatisfactory as they only cover a finite number of cases when the problem is infinite in two dimensions (regular expressions and input words). Random testing seems more appropriate: it can generate random expressions or it can draw example expressions from online repositories with reams of real life regular expressions [RegExLib [n. d.]], but then there needs to be an oracle for the generated language and it is non-trivial to define generators for test inputs.

We eliminate the oracle by providing generators for positive and negative example words. Generators for positive examples, which match a given regular expression, have been investigated [Ackerman and Shallit 2009; Groz and Maneth 2017; Mäkinen 1997; McIlroy 2004], although mainly in a theoretical context. Generating negative examples, which do **not** match an expression, has not been considered before.

The generators presented in this work are yet more general. They apply to **extended regular expressions** that

contain intersection and complement beyond the standard 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 algorithms produce lazy streams, which are guaranteed to be productive (i.e., their next element is computed in finite time). A user can limit the string length or the number of generated strings without risking partiality.

Source code for implementations in Haskell[1] and in OCaml[2] is available on GitHub. Examples can be run in a Web App[3]. Although 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.

§ 2–5 investigate the overall design using Haskell. Fine tuning of the underlying data structures is investigated using OCaml in § 6. § 7 reports benchmarking results, § 8 considers testing, § 9 discusses related work, and § 10 concludes.

We assume fluency with Haskell and OCaml. Familiarity with formal languages is helpful, but not required as the paper contains all relevant definitions.

2 Motivation

Suppose someone implemented a clever algorithm for regular expression matching, say `match`. We want to use this implementation, but we also want to make sure it is largely bug free by subjecting it to extensive testing. So we need 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 `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`.

QuickCheck [Claessen and Hughes 2000] is a library for property-based random testing, which is well suited for conducting such a test. Using QuickCheck, we would write a generator for regular expressions and then use the generator for strings to generate many inputs for each expression.

However, this approach has a catch. Depending on the language of the regular expression, the probability that a uniformly distributed 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.

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

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

[1] https://github.com/peterthiemann/re-generate
[2] https://github.com/regex-generate/regenerate
[3] https://regex-generate.github.io/regenerate/

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

Hence, there are two problems.

1. How do we know whether the test oracle is correct, short of verifying it?
2. How do we generate relevant test cases, given that the density of many regular languages is 0 or 1?

Wouldn't it be nice to have a systematic means of generating words **inside** of L and **outside** of L? Such a generation algorithm would eliminate the oracle and it would give us control over the number of test inputs in the language and in the language's complement.

To construct such an algorithm we tackle the more general question of generating the language of a regular expression extended with operators for intersection (&) and complement (~). This algorithm can generate the complement of $L(r)$ by asking it to generate $L(\sim r)$.

Requirements For the testing application, we propose some requirements for the generation algorithm to avoid inefficiencies and spurious testing failures.

1. No repetitions in the output.
2. Output must not be partial.
3. Throttling output with respect to word length and number of generated words should be possible.
4. Generation should be compositional.
5. Reasonable efficiency.

3 Previous Work

3.1 Brief Intermezzo on Formal Languages

Let Σ be a finite set, the **alphabet**, totally ordered by $<$. We write Σ^* for the set of finite words over Σ, which is defined by $\bigcup_{i=0}^{\infty} \Sigma^i$ where $\Sigma^0 = \{\varepsilon\}$ and $\Sigma^{i+1} = \Sigma \times \Sigma^i$. The semantics of a regular expression, $[\![r]\!] \subseteq \Sigma^*$, is a set of words, defined in Figure 1. 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 $U^{i+i} = U \cdot U^i$.

An extended regular expression (Figure 1) is 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, 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 expressions can be more concise.

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

```
data GRE sig
  = Zero
  | One
  | Atom sig
  | Or (GRE sig) (GRE sig)
  | Dot (GRE sig) (GRE sig)
  | Star (GRE sig)
  | And (GRE sig) (GRE sig)
  | Not (GRE sig)
```

Figure 1. Generalized regular expressions: syntax, semantics, and Haskell encoding

3.2 McIlroy's Approach

McIlroy [2004] enumerates the words of a regular language as a strictly increasingly ordered infinite stream using Haskell. A key insight is to use the length-lexicographic ordering on words, which is defined by $u \leq_{ll} v$ if $|u| < |v|$ or $|u| = |v|$ and $u \leq v$ in the usual lexicographic ordering. Here is a definition in Haskell.[4]

```
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 gives raise to an enumeration of all words over the alphabet Σ via an order-preserving bijection from the natural numbers to Σ^*. Using this bijection we can show that for each pair of words $v \leq_{ll} w$ the number of words u such that $v \leq_{ll} u$ and $u \leq_{ll} w$ is finite.

The Haskell code below defines a generator as a compositional function in terms of the regular operations.

```
import qualified Data.Text as T

type Alphabet = [Char] -- ascendingly sorted
type Lang = [T.Text]   -- lazy stream of words

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)
```

[4]The type `Data.Text.Text`, commonly imported as `T.Text`, is an efficient string data type from the Haskell library. We only use `T.empty` for the empty string, `T.append` for string append, `T.singleton` to create a single letter word, and `T.length`.

```
module Examples.McIlroy where
import qualified Data.Text as T
import Examples.LLO (llocompare)

union :: Lang -> Lang -> Lang
union xs@(x:xs') ys@(y:ys') =
  case llocompare x y of
    EQ -> x : union xs' ys'
    LT -> x : union xs' ys
    GT -> y : union xs ys'
union xs ys = xs ++ ys

concatenate :: Lang -> Lang -> Lang
concatenate [] ly = []
concatenate lx [] = []
concatenate (x:xt) ly@(y:yt) =
  T.append x y : union (concatenate [x] yt)
                       (concatenate xt ly)

star :: Lang -> Lang
star [] = [T.empty]
star lx@(x:xt) = if x == T.empty
  then star xt
  else T.empty : concatenate lx (star lx)
```

Figure 2. McIlroy's implementation of regular operators

```
    gen (Not r) = complement sigma (gen r)
```

Figure 2 contains McIlroy's implementation of the regular operators. His definitions for `concatenate` and `star` are more general, but that generality is not needed for this application.

The `union` operation is implemented as a merge of two strictly increasing streams; the last line only applies if one of the streams is empty. This definition is productive and yields a strictly increasing stream.

The `concatenate` operation is generic sequence multiplication, which is productive. It yields a strictly increasing stream because `T.append x y` is the smallest element in the product of `lx` and `ly` if `x` is smallest in `lx` and `y` is smallest in

```haskell
-- continuing module Examples.McIlroy

intersect :: Lang -> Lang -> Lang
intersect xs@(x:xs') ys@(y:ys') =
  case llocompare x y of
    EQ -> x : intersect xs' ys'
    LT -> intersect xs' ys
    GT -> intersect xs ys'
intersect xs ys = []

-- difference :: Lang -> Lang -> Lang
-- omitted for space reasons

complement :: Alphabet -> Lang -> Lang
complement sigma lx = difference lsigmastar lx
  where
    lsigmastar = star (map T.singleton sigma)
```

Figure 3. Additional operations in McIlroy's framework

`ly`. If one of the input languages is infinite, then there is a `union` operation for each generated word.

The `star` operation is defined by closure under concatenation. The definition is productive and the output is strictly increasing as it is generated by `concatenate`.

This implementation is correct by construction as it closely follows the mathematical definition of the operations. However, McIlroy readily admits in the paper it is very inefficient.

3.3 Extending McIlroy

McIlroy's paper does not mention that the same representation of a language enables the efficient implementation of `intersect`, `difference`, and hence the `complement` operations!

Figure 3 shows that intersection and difference can be implemented as simple variations of the `union` operation. They run in linear time on finite lists. Moreover, given a stream corresponding to Σ^*, which is easy to define, and the difference operation, we obtain a definition of `complement`.

At this point, we have an implementation of all operations, but concatenation (and hence star) is inefficient. Observe further that neither `intersect` nor `difference` are productive: `intersect` applied to two eventually disjoint infinite streams is partial. For example, computing the intersection $(ab)^* \cap (ba)^*$ yields a partial list, which starts with the empty word, but never produces another word. As another example, computing the difference $(aa)^* \setminus a^*$ never produces any word. Hence, the `complement` operation is not productive, either.

These definitions are not acceptable because they may produce spurious test failures due to timeouts. Moreover, they make it impossible to reliably gauge test inputs by length or number, because the generator may get stuck in an unproductive loop before reaching the proposed limit.

```haskell
concatenate :: SegLang -> SegLang -> SegLang
concatenate lx ly = collect 0
  where
    collect n =
      (foldr McIlroy.union []
         $ map (combine n) [0 .. n])
      : collect (n+1)
    combine n i =
      liftA2 T.append (lx !! i) (ly !! (n - i))
```

Figure 4. Concatenation for segment representation

4 Generation by Cross Section

We address the problems outlined with McIlroy's approach by switching to a different **segment representation** for the generated language.

Let $L \subseteq \Sigma^*$ be a language and n be a natural number. The n^{th} **cross section of** L or n^{th} **segment of** L is $L_n := L \cap \Sigma^n$, the set of words of length n in L. As $L = \bigcup_{n \geq 0} L_n$ we define the **segment representation of** L by the sequence of all segments $(L_n)_{n \geq 0}$.

The language operations can be expressed on this representation by standard operations on sequences.

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

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

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

In a language that supports streams, all operations become executable. As the alphabet Σ is finite, each segment L_n is a finite set. Hence, the sum and Hadamard product yield efficient definitions of language union and intersection. Due to the change of representation, the intersection is productive: it produces the next segment in finite time.

The Haskell representation of a language has the type

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

where the outer list is assumed to be an infinite lazy stream and each inner list is a finite, strictly increasing list of words of the same length. On words of the same length, the length-lexicographic order is the same as the lexicographic order.

The union, intersection, and difference operators on `SegLang` are defined according to Equations (1) and (2).

```haskell
union = zipWith McIlroy.union
intersect = zipWith McIlroy.intersect
difference = zipWith McIlroy.difference
```

Concatenation As all words in $U_i \cdot V_{n-i}$ in Equation (3) have length n, they belong to the n^{th} segment of the result. Both U_i and V_{n-i} are strictly increasingly sorted, so the standard enumeration of pairs from $U_i \times V_{n-i}$ is sorted in the same way. Hence, our implementation of `concatenate` in

```
star :: SegLang -> SegLang
star lx = lstar
  where
    lstar = [T.empty] : collect 1
    collect n =
      (foldr union []
         $ map (combine n) [1 .. n])
      : collect (n + 1)
    combine n i =
      liftA2 T.append (lx !! i) (lstar !! (n - i))
```

Figure 5. Kleene closure for segment representation

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

Figure 6. Complementation for the segment representation

Figure 4. Function `combine` implements $U_i \cdot V_{n-i}$ and `collect n` computes the stream of segments starting from the n^{th}. Expression `lx !! i` accesses the `i`th element of list `lx`.

Kleene Closure It is well known that $U^* = (U \setminus \{\varepsilon\})^*$. Hence, a simple calculation yields an effective algorithm for computing the sequence of cross sections for U^*.

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

The key observation is that Equation (4) is a proper inductive definition of the sequence for U^*. It never touches U_0 and the union only touches the elements $(U^*)_{n-1}$ down to $(U^*)_0$. Hence, $(U^*)_n$ is well defined as it only relies on U and previously computed indexes!

Figure 5 contains the resulting implementation of Kleene closure. The `collect` and `combine` functions are almost identical, `lstar` is defined recursively, but this recursion is well-founded as we just argued.

Complement To define the `complement` operation, all we need is to define the segment representation of Σ^*, which can be done analogously to computing the closure, and apply the `difference` operator. Figure 6 puts it all together.

Discussion What have we gained?

Productivity: We can generate productive segment representations from all extended regular expressions. The implementation of each operation is guided by corresponding operations on streams.

Easy Gauging: To restrict the generated segmented output, say `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 contrast, such filtering is not effective for the LLO representation where `takeWhile (\w -> T.length w < n) llo` may yield a partial list.

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

Figure 7. Concatenation with convolution

There is a catch: As the output is always an infinite list, we lose the information that a language is finite. The advantange of this choice is simplicity: index accesses into the languages `lx` and `ly` are always defined. However, the list index operation used in the `combine` function requires time linear in its index argument, which may be inefficient. The next section discusses ways to address these shortcomings.

5 Improvements

5.1 Faster Concatenation by Convolution

Looking back at the code in Figure 4, 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.

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 7.

The code in the figure only handles infinite lists of segments. When extended with handling the case for finite lists, it gives rise to an 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.

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

Figure 8. Base cases for refined segment representation

We implemented the finiteness-aware algorithm including role flipping, but we do not include the code in the paper, as it is a tedious variation of the code in Figure 7.

5.2 Refined Segment Representation

The refined segment representation attempts to represent finite languages by finite segment sequences as much as possible. Hence, the base cases for the interpretation of a regular expression may be defined as in Figure 8. The implementation of `union`, `intersect`, `difference`, and `complement` is an easy exercise and thus omitted. We discuss the remaining issues with the previous code for concatenation in Figure 7.

Function `concatenate''` always returns an infinite list of segments. To keep that list finite, we need a termination criterion for `collect` to determine that all future calls will only contribute empty segments.

To this end, we 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 5.1. *Let $X, Y \subseteq \Sigma^*$ be languages and suppose that there are numbers $m_x, m_y \geq 0$ such that*

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

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`. We omit the straightforward modification of the code.

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.

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 still an infinite list of empty segments.

§ 5.4 discusses ways to get finite representations from more languages. Generally it is impossible to guarantee a finite language is represented by a finite stream of segments.

5.3 Faster Closure

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

5.4 More Finite Representations

We already remarked that we can keep segmented representations finite for finite languages constructed without using `complement`. To extend the realm of finite representations we propose to use a custom data types for segment lists.

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

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

Constructor `Empty` represents the empty set. A segment list `Cons xl xsegs` represents the union of the language `xl` and segments `xsegs`. If 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 data type `Lang` to represent languages $L \subseteq \Sigma^n$, for some n. 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 Σ^n.

It is an easy 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:[5]

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

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

We can extend the range of languages with finite representations by dualizing the idea for detecting finiteness in § 5.2. The following lemma captures this idea.

Lemma 5.2. *Let $X, Y \subseteq \Sigma^*$ be languages and suppose that there are numbers $f_x, f_y \geq 0$ such that*

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

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

The generation algorithm can determine f_x and f_y by detecting when a segment list is `Full`. Once f_x and f_y are

[5] We use string notation for elements of `Data.Text.Text` for readability. The bar in `Full _` stands for an unevaluated infinite list of full segments.

both determined and the generated word length n is greater than or equal to $f_x + f_y$, then we can finish producing the segments by outputting the appropriate `Full`.

This approach yields finite segment representations for many finite and co-finite languages. But the algorithm is easy to defeat. For example, both $[\![a^* + \sim(a^*)]\!] = \Sigma^*$ and $[\![a^* \,\&\, \sim(a^*)]\!] = \emptyset$ are both mapped to infinite segment lists.

6 OCaml Implementation

We also implemented our language generation algorithm in OCaml. 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 stream operations. To facilitate such experimentation, we implemented the algorithm as a functor whose signature is shown in Figures 9 to 11. OCaml functors are parameterized modules that take modules as argument and return modules. Our implementation takes two data structures as arguments, words and segments, to test different representations without changing the code.

Characters and Words Figure 9 contains the signature for words. It provides the empty word (for `One`), singleton words (for `Atom`), and append. Neither an ordering nor a length operation is needed: 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 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 10 contains the signature for segments. The main requirement is to support the operations on power series as described in section 4 and the set operations `union`, `inter` and `inter`. The product described in Equation 3 is decomposed in two parts:

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

Experimentation with transient data-structures requires an explicit `memoize` function that avoids recomputing segments accessed multiple times. Finally, the functions `of_list` and `iter` import and export elements to and from a segment.

6.1 Core Algorithm

The core algorithm follows the Haskell version. The power series is implemented using a thunk list in the style of Pottier [2017] with some special-purpose additions:

```
type node =
  | Nothing
  | Everything
  | Cons of Segment.t * lang
and lang = unit -> node
```

An enumeration is represented by a function which takes a unit argument and returns a node. A node, in turn, is either `Nothing` or a `Cons` of an element and the tail of the sequence. The additional constructor `Everything` allow to manipulate full languages symbolically, as in § 5.4.

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

Figure 9. Operations on words

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

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

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

  (* Import/Export *)
  val of_list: elt list -> t
  val iter: t -> (elt -> unit) -> unit

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

Figure 10. Operations on segments

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

Figure 11. Language generation as a functor

```
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
```

Figure 12. Signature for `ThunkList`

As an example, the implementation of language union is shown below. The trailing unit argument `()` drive the evaluation of the sequence lazily. With this definition, `union s1 s2` cause no evaluation before it is applied to `()`.

```
let rec union s1 s2 () = match s1(), s2() with
  | Nothing, x | x, Nothing -> x
  | Everything, _ | _, Everything -> Everything
  | Cons (x1, next1), Cons (x2, next2) ->
    Cons (Segment.union x1 x2, union next1 next2)
```

6.2 Data Structures

Our parameterized implementation enables experimentation with various data structures for segments. We present several possibilities before comparing their performance.

Ordered enumerations Ordered enumerations, represented by thunk-lists, make for a light-weight set representation. To use an order, we require `compare` and `append` on words. The `OrderedMonoid` signature captures these requirements. Figure 12 shows the resulting functor `ThunkList`.

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

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 using a growing vector as cache. Such a memoization function incurs a linear cost on enumerations. To test if this operation is worthwhile we implemented two modules: `ThunkList` without memoization and `ThunkListMemo` with the implementation described above.

Lazy Lists OCaml also supports regular lazy lists using the builtin `Lazy.t` type. We implemented a `LazyList` functor which is identical to `ThunkList` but uses lazy lists.

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 which relies on balanced binary trees. The only operations not implemented by OCaml's standard library are the n-way merge and the product.

Tries Tries [Fredkin 1960] are prefix trees where each branch is labeled with a character and each node may contain a value. Tries are commonly used as maps from words to values where 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: since all words in a segment have the same length, we only need values at the leaves. Hence, we can implement tries like tries of integers [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 either `Empty`, a `Leaf` or a `Node` containing a map from characters to its child tries. The only novel operation is `append` which computes the product of two sets. It can be implemented in a single traversal which grafts the appended trie `t0` at each leaf of `t`, without copies.

```
type trie = Empty | Leaf | Node of trie CharMap.t
let rec append t t0 = match t with
  | Empty -> Empty | Leaf -> t0
  | Node map ->
    CharMap.map (fun t' -> append t' t0) map
```

7 Benchmarks

We consider the performance of our implementations in two dimensions: first the successive algorithmic refinements in the Haskell implementation presented in § 4 and 5, then the various segment representations in OCaml as described in § 6.

Benchmarks were executed 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.

7.1 Comparing Algorithms in the Haskell Implementation

§ 4 and 5 develop the algorithm for generating languages in a sequence of changes applied to a baseline algorithm. We evaluate the impact of these changes on performance by measuring the generation speed in words per second. This speed depends heavily on the particular regular expression. Thus, we select four 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 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.
- $\sim(a^*) \cdot b$: 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.
- $\sim(a^*)$ & $\sim(b^*)$: This regular expression applies intersection to two large languages and make use of the complement. Its goal is to measure the efficiency of set operations.

We consider five variants of the Haskell implementation.

- **McIlroy** our implementation of McIlroy [2004].
- The **seg** implementation uses the infinite list-based segmented representation throughout (§ 5.2).
- The **segConv** implementation additionally applies the convolution approach (§ 5.1 and 5.3).
- The **refConv** implementation combines symbolic segments (§ 5.2 and 5.4) with the convolution approach.

Performance is evaluated by iterating through the stream of words produced by the generator, forcing their evaluation[6] and recording the elapsed timed every 20 words for 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 indicates the generation speed of the algorithm, high slope is correlated to high generation speed. Figure 13 contains the results for the Haskell implementations.

Most algorithms generate between $1.3 \cdot 10^3$ and $1.4 \cdot 10^6$ words in the first second, which seems sufficient for testing purposes. The **refConv** implementation which uses symbolic segments and convolutions is consistently in the leading group. This observation validates that the changes proposed in § 5 actually lead to improvements. Looking at each graph in detail, we can make the following remarks:

- All implementations are equally fast on a^* except **McIlroy**, which implements star inefficiently.
- 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.
- The expression $(a \cdot b^*)^*$ demonstrates that the convolution technique presented in § 5.1 leads to significant improvements when applying `star` to non-sparse languages.

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

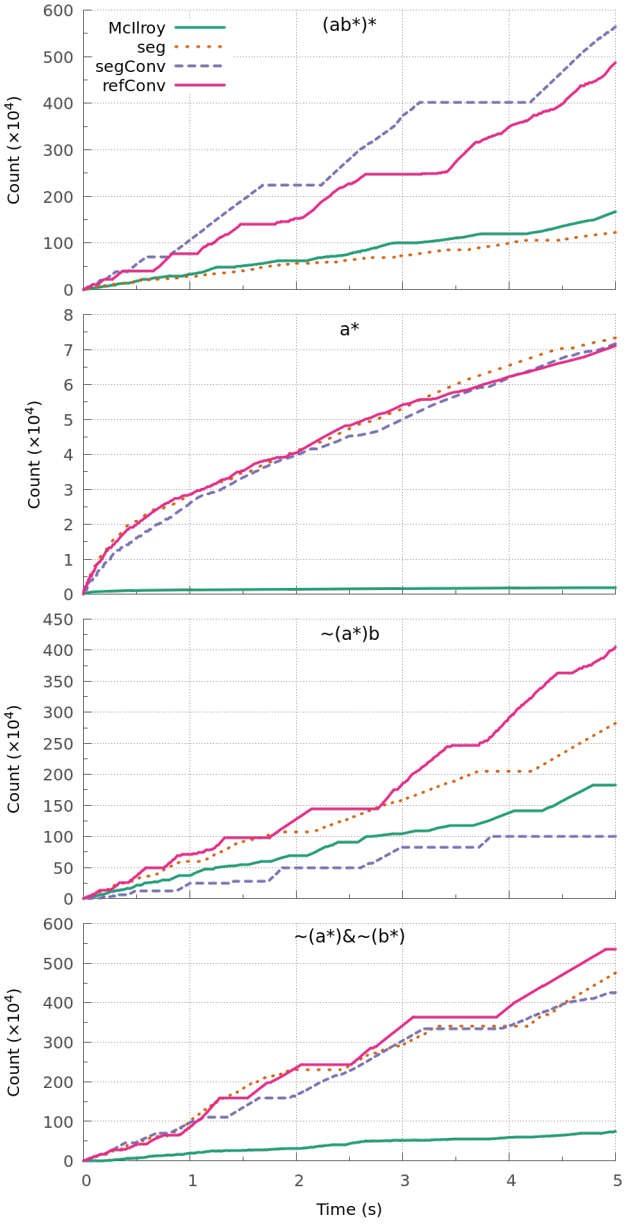

Figure 13. Benchmark for the Haskell implementation with various algorithms

- The **refConv** algorithm is 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**.
- The expression $\sim(a^*)$ & $\sim(b^*)$ shows that all our algorithm have similar performance profiles on set-operation. They are also significantly faster than **McIlroy**.

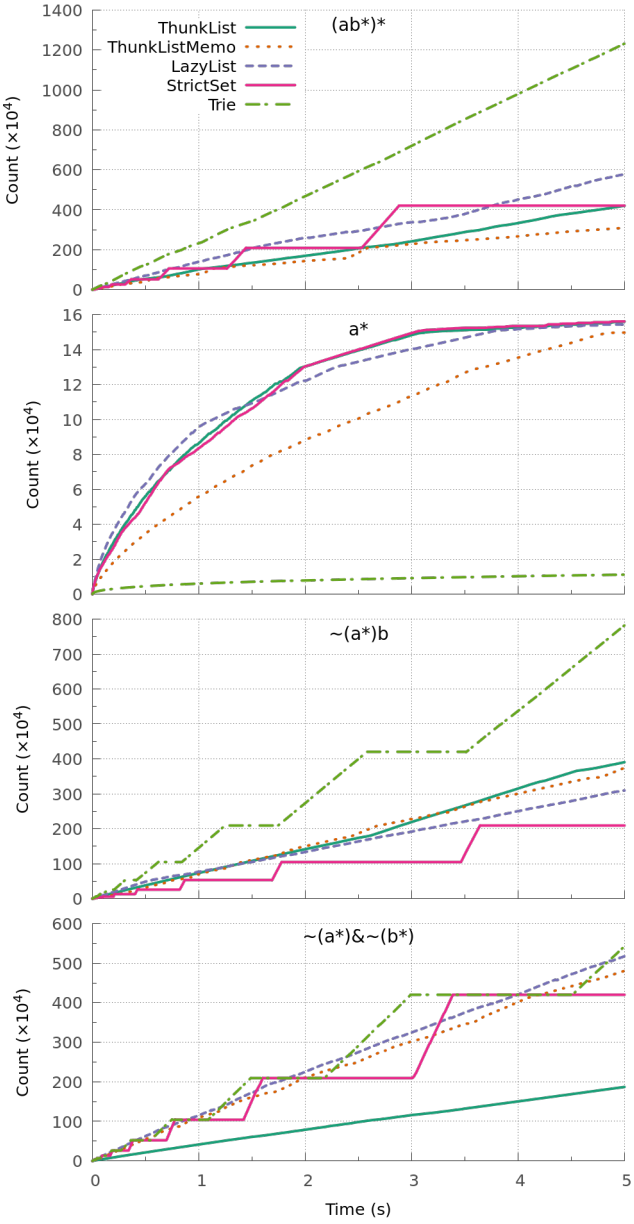

Figure 14. Benchmark for the OCaml implementation with various data-structures

7.2 Comparing Data Structures in the OCaml Implementation

We have now established that the **refConv** algorithm provides the best overall performance. The Haskell implementation, however, uses lazy lists to represent segments. To measure the influence of strictness and data structures on performances, we conduct experiments with the functorized OCaml implementation. 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 14.

Unlike the previous benchmark for algorithms, there is no clear winner among the data structures. Lazy and Thunk lists, with or without memoizations, are the most "well-rounded" implementations and perform decently on most languages.

- The `Trie` module is very fast thanks to its efficient concatenation. It performs badly on a^* due to the lack of 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 compression 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. Far less memory is consumed for other languages.
- Strict data structures showcase a marked "skewed stair" pattern, which is completely absent for `ThunkList` and `LazyList`. Thus, manual control of laziness works 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 does not significantly improve performance. It seems that the linear cost of memoizing the thunk list and allocating the vectors is often higher than simply recomputing the lists.
- The expression $(a \cdot b^*)^*$ shows that sorted enumerations and tries perform well on set-operations, even compared to strict sets.

7.3 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 `LazyList` OCaml implementation. Before presenting the results, a word of warning: We do not claim to offer a fair comparison between languages! The two implementations are not exactly the same and we made no attempt to measure both languages under exactly the same conditions. Figure 15 contains the results with 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.
- $(1(01^*0)^*1 + 0)^*$, the language of multiples of 3 in binary representation. Again, this is a language that is neither finite nor cofinite, but its segments are never full nor empty.

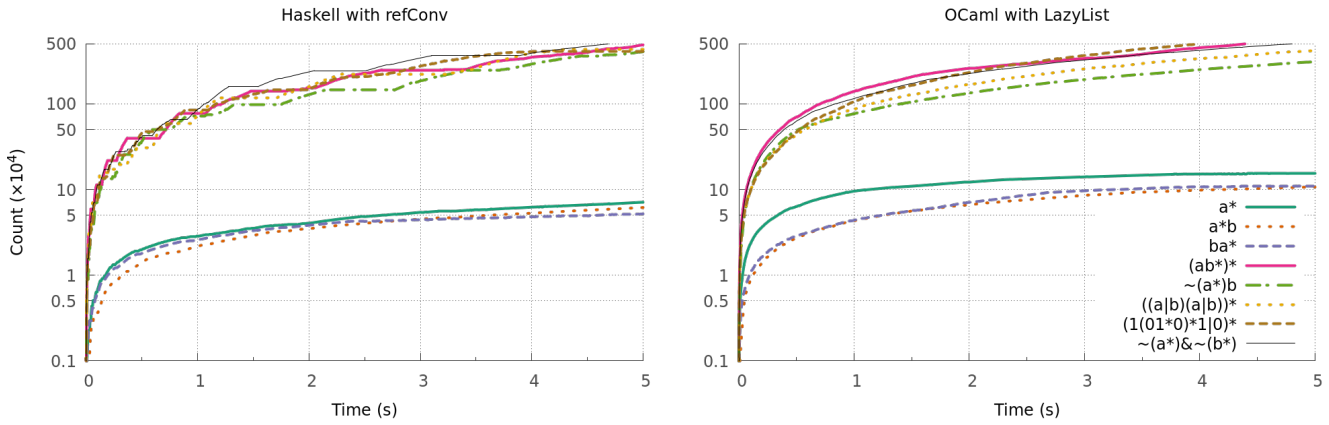

Figure 15. Benchmark on different regular expressions

- a^*b and ba^*, which together check whether the performance of concatenation is symmetric.

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 both implementations, thanks to the improved convolution technique with detection of finite languages described in § 5.1.

8 Testing

We implemented our algorithms in a library to implement a test harness for the OCaml Re library[7], a commonly used OCaml regular expression implementation. We also created a set of test cases for student projects in Haskell to help them write better implementations.

Both libraries provide test harnesses which generate regular expressions with positive and negative samples. The implementation under test compiles the regular expression and applies it to the samples. The library exposes a sample generator in the style of property testing libraries such as QuickCheck [Claessen and Hughes 2000]. This way we can use the tooling already available in such libraries. The simplified API of the OCaml version is shown below. The main function `arbitrary n alphabet` returns a generator which provides on average `n` samples using the given alphabet.

```
type test = {
  re : Regex.t ;
  pos : Word.t list ;
  neg : Word.t list ;
}
val arbitrary:
  int -> Word.char list -> test QCheck.arbitrary
```

[7] https://github.com/ocaml/ocaml-re

Regular expressions are easy to generate using QuickCheck-like libraries as they are represented by an algebraic datatype. We restrict generated regular expressions to star-heights less than 3. While our technique can be used for regular expressions with arbitrarily nested repetitions, it can cause slowdown and large memory consumption which are inconvenient in the context of automated testing.

Our library returns a finite number of samples even for an infinite language. We want to generate test-cases that exercise the implementation under test as much as possible. For this purpose, we use a technique similar to the fast approximation for reservoir sampling [Vitter 1987]. When considering the sequence of words in the language, we skip k elements where k follows a power law of mean n. We then return the given sample, and stop the sampling with a probability $1/n$. This technique results on average in k samples that are regularly spaced at the beginning of the stream, but will occasionally skip ahead and return very long words. This approach has proven satisfactory at finding good testing samples in practice.

9 Related Work

Regular Language Generation

Mäkinen [1997] describes a method to enumerate the words of a regular language L, given by a deterministic finite automaton, in length-lexicographic ordering. To generate words up to length n, this method precomputes in time $O(n)$, for each $i \leq n$, the lexicographically minimal and maximal word of length i in L. Enumeration starts with the minimal word of length n and repeatedly computes the lexicographically next word in L until it reaches the maximal word of length n. Each step requires time $O(n)$.

In comparison, Mäkinen requires a deterministic finite automaton, which can be obtained from a regular expression in worst-case exponential time. Complementation is not mentioned, but it could be handled. Mäkinen would give rise to a productive definition by cross sections because the computation of minimal and maximal words could be done incrementally, but which is not mentioned in the paper.

McIlroy [2004] implements the enumeration of all strings of a regular language in Haskell. He develops two approaches, one based on interpreting regular expressions inspired by Misra [2000] and discussed in Section 3.2, the other (unrelated to ours) using a shallow embedding of nondeterministic finite automata.

Ackerman and Shallit [2009] improve Mäkinen's algorithm by working on a nondeterministic finite automaton and by proposing faster algorithms to compute minimal words of a given length and to proceed to the next word of same length. An empirical study compares a number of variations of the enumeration algorithm. Ackerman and Mäkinen [2009] present three further improvements on their enumeration algorithms with better asymptotic complexity. The improved algorithms perform better in practice, too.

Ackerman's approach and its subsequent improvement does not incur an exponential blowup when converting from a regular expression. As it is based on nondeterministic finite automata, complementation cannot readily be supported. Moreover, the approach is not compositional.

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 a 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 § 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] enumerate elements of various data structures. Their approach is complementary to test-data generators. It exploits bijections between natural numbers and the data domain and develops a quality criterion for data generators based on fairness.

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.

10 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 then 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 § 7, 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

Margareta Ackerman and Erkki Mäkinen. 2009. Three New Algorithms for Regular Language Enumeration. In *COCOON (Lecture Notes in Computer Science)*, Vol. 5609. Springer, 178–191.

Margareta Ackerman and Jeffrey Shallit. 2009. Efficient Enumeration of Words in Regular Languages. *Theor. Comput. Sci.* 410, 37 (2009), 3461–3470.

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.

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

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

Erkki Mäkinen. 1997. On Lexicographic Enumeration of Regular and Context-Free Languages. *Acta Cybern.* 13, 1 (1997), 55–61.

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/.

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

Jeffrey Scott Vitter. 1987. An efficient algorithm for sequential random sampling. *ACM Trans. Math. Softw.* 13, 1 (1987), 58–67. https://doi.org/10.1145/23002.23003

M. Zalewski. 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