# Efficient Incremental Evaluation of Succinct Regular Expressions

Henrik Björklund
Umeå University

Wim Martens *
Universität Bayreuth

Thomas Timm
Universität Bayreuth

## ABSTRACT

Regular expressions are omnipresent in database applications. They form the structural core of schema languages for XML, they are a fundamental ingredient for navigational queries in graph databases, and are being considered in languages for upcoming technologies such as schema- and transformation languages for tabular data on the Web. In this paper we study the usage and effectiveness of the counting operator (or: limited repetition) in regular expressions. The counting operator is a popular extension which is part of the POSIX standard and therefore also present in regular expressions in grep, Java, Python, Perl, and Ruby. In a database context, expressions with counting appear in XML Schema and languages for querying graphs such as SPARQL 1.1 and Cypher.

We first present a practical study that suggests that counters are extensively used in practice. We then investigate evaluation methods for such expressions and develop a new algorithm for efficient incremental evaluation. Finally, we conduct an extensive benchmark study that shows that exploiting counting operators can lead to speed-ups of several orders of magnitude in a wide range of settings: normal and incremental evaluation on synthetic and real expressions.

## 1. INTRODUCTION

Regular expressions are omnipresent in programming languages, shell tools, and database query- and schema languages. In the context of XML, DTDs and XML Schema definitions use regular expressions for defining content models and XPath uses regular-expression-like constructs for navigation in trees. In the context of graph-structured data and RDF, regular expressions have been studied under the name *regular path queries (RPQs)* for decades and have recently been integrated in popular graph query languages such as SPARQL [16] and Cypher [24]. Currently, they are being considered as a basic building block for processing tabular data on the Web [22].

We investigate the usage and effectiveness of regular expressions with the counter operator, also known as the limited repetition operator. This operator makes some expressions exponentially more succinct and permits a subexpression to be matched anywhere between a given *minimum* and *maximum* number of times. For example, the expression `(ab){9,20}` (in POSIX syntax) matches all words that consist of at least nine and at most twenty repetitions of `ab`. Expressions with counting are relevant to several authoritative languages for processing XML and RDF. In particular, they form the structural core of XML Schema Definitions,[1] where they are used to define content models [10]. In the context of RDF and graph query languages, counting operators are used in Cypher's *path patterns* [24] and they have been considered in SPARQL 1.1 *property paths* [16]. However, the effect of counters in regular expressions is not yet fully understood, which is why they were removed from SPARQL 1.1. For this reason we want to contribute to the understanding of such succinct regular expressions and show that (1) counting operators are widely used in practice and therefore seem desirable to practitioners and (2) can be exploited to develop highly efficient evaluation algorithms.

We start with a practical study that investigates how counters are used in regular expressions in practice. We investigate *RegExLib*, the main regular expression repository available on the Web [26]; *Snort*, a system for detecting network intrusion [28], and a large corpus of XML Schema definitions which we harvested from the Web. The data suggests that counters are used often and that counter values can become rather large. For example, in XML Schema we found values up to ten million. We also discovered that schemas of companies such as Amazon, Ebay, Microsoft, Oracle, and Paypal use expressions with counting.

We then turn to efficient evaluation of expressions. The "standard" evaluation problem for expressions is defined as: Given an expression $e$ and a word $w$, can $e$ be matched onto $w$? However, many settings in databases call for a more refined version of evaluation. For example, the word $w$ is usually not static and is subject to updates. Examples of such settings are (a) schema constraints and integrity constraints that safeguard the validity of the data, and (b) trigger conditions that continuously monitor whether some action needs to be performed. In settings like these, one would like to perform *incremental evaluation*. Here, we start with a word

*Supported by grant number MA 4938/2–1 from the Deutsche Forschungsgemeinschaft (Emmy Noether Nachwuchsgruppe).


ACM 978-1-4503-3794-6/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2806416.2806434 .

[1]The counting operators are called minOccurs/maxOccurs in XML Schema.

$w$ and an expression $e$ that matches $w$, and then the word $w$ is updated (see, e.g., [2, 3, 7]). We want to find out quickly if the expression still matches the updated word $u(w)$. In order to do so, incremental evaluation algorithms may store *auxiliary data*. Naturally, we are interested in striking a good balance between the speed of handling the update and the size of the auxiliary data.

We develop an incremental evaluation algorithm for regular expressions with counting and show that it is possible to avoid the usual exponential translation of such expressions to (standard) finite automata which is used by e.g., current implementations of tools like grep.

We then perform an extensive experimental study that evaluates the efficiency of our evaluation algorithms on synthetic and real-world expressions. As expected, avoiding the exponential blow-up to automata leads to significant speed-ups. When expressions have large counter values, we see speed-ups of several orders of magnitude in every setting we investigate. We look into both normal and incremental evaluation, on both synthetic and real-world expressions.

**Related work.** Although incremental evaluation of queries and automata has attracted much attention, this is the first work on incremental evaluation of regular expressions with counters, to the best of our knowledge. In the context of databases, the most well studied problem in this context is *incremental view maintenance*. However, in contrast to that problem, where one aims at maintaining the (possibly large) output of a query, we only want to decide whether a query or a pattern can be matched or not. In this sense, the present work is much closer to incremental evaluation of XML schemas [2, 3] and the matching of XPath patterns [7]; a line of work going back to [25].

Efficient evaluation of regular expressions is also heavily investigated in deep packet inspection in networking (see, e.g., [4, 5]). In this context, fast evaluation of regular expressions with counting has been investigated by Smith et al. [27]. While this line of work bears similarities to ours, the focus is quite different. Most prominently, the focus is on evaluating the expressions on a *data stream*. Our setting is more general because we allow insertions and deletions in arbitrary positions of the word, while a data stream is essentially a word which only receives insertions at the back.

Ghelli et al. [13] study fast evaluation for a class of regular expression with counting and interleaving. In the expressions in their study, each alphabet symbol can appear at most once and iteration (counters and Kleene star) can only be applied directly to alphabet symbols. They develop a linear-time algorithm but it does not work for all regular languages, whereas ours does. It would be interesting to investigate if the advantages of both settings can be combined. Another setting where regular expressions are augmented with intervals was studied by Nakayama et al. [23], who also provide an efficient evaluation algorithm. There, however, the intervals do not signify repetition, but rather duration and the strings considered are ones where each letter has an assigned duration. Our incremental evaluation algorithm is designed for *automata with counters*, see, e.g., [18, 9].

## 2. EXPRESSIONS WITH COUNTING

The *regular expressions* over a set of symbols $\Sigma$, denoted by RE, are defined as follows: $\varepsilon$, $\emptyset$, and every $\Sigma$-symbol is in RE; and whenever $r$ and $s$ are in RE, then so are $(rs)$, $(r+s)$, and $(r^*)$. For readability, we omit parentheses where appropriate. The language defined by an expression $r$ is denoted by $L(r)$ and defined as usual. We denote the regular expressions with counters as $\mathrm{RE}^{\#}$ and define them as follows. Every RE-expression is an $\mathrm{RE}^{\#}$-expression. Furthermore, when $r$ is an $\mathrm{RE}^{\#}$-expression then so is $r^{k,\ell}$ for $k \in \mathbb{N}$ and $\ell \in \mathbb{N}^+ \cup \{\infty\}$ with $k \leq \ell$. Here, $\mathbb{N}^+$ denotes $\mathbb{N} \setminus \{0\}$. Furthermore, $L(r^{k,\ell}) = \bigcup_{i=k}^{\ell} (L(r))^i$, where $(L(r))^i$ denotes the $i$-fold concatenation (or repetition) of $L(r)$. Notice that counters are just syntactic sugar. An expression $r \in \mathrm{RE}^{\#}$ can always be converted to an equivalent RE by "unfolding" the counters. The blow-up is worst-case exponential.

## 3. A PRACTICAL STUDY

We conducted an extensive practical study to get an idea of how counters are used in regular expressions. We investigated expressions in the RegExLib repository [26], in Snort rules [28], and conducted a thorough search for expressions with counters in XML Schemas on the Web.

In the remainder of this section, we say that an expression uses *non-trivial counters* if it has a subexpression of the form $r^{k,\ell}$ in which one of $k$ or $\ell$ is at least two, i.e., in $\mathbb{N} - \{0, 1\}$.

In a nutshell, we discovered that RegExLib and Snort rules have a surprisingly large fraction (>50%) of expressions with non-trivial counters. RegExLib and Snort have expressions with fairly large counter values (between 1K and 10K), but the largest counters we found occurred in XML Schemas and were up to ten million.

### 3.1 RegExLib

The RegExLib.com library describes itself as the Internet's first Regular Expression Library [26]. It has a corpus of expressions for recognising URIs, markup code, pieces of Java code, SQL queries, spam, etc. We crawled the expressions of the library and obtained a set of 3024 expressions after removing duplicates and expressions that could not be parsed by Bart Kiers' PCREParser available at Github.[2]

Of these 3024 expressions, 1705 (about 56.3%) use non-trivial counters. This seems to indicate that counters are widely used in practice. We investigated the size of the counters and the nesting of non-trivial counters so that we could get an idea of how large equivalent expressions *without* counters would become. This estimates the internal blow-up that happens in tools like grep. For subexpressions of the form $r^{k,\ell}$ in the library, we found values of $k$ ranging from 0 to 255 and values of $\ell$ ranging from 1 to 1500.

However, some expressions use nesting of counters, which lead to even larger minimal equivalent REs. We discovered that 86 expressions in the corpus use nesting of non-trivial counters. Several expressions in the corpus contain nested counters of the form $(r_3(r_2(r_1^{k_1,\ell_1}))^{k_2,\ell_2})^{k_3,\ell_3}$ where $\ell_1 = 12$, $\ell_2 = 255$, and $\ell_3 = 255$. The smallest equivalent RE would require at least 780,000 symbols. Another example uses a four-fold nesting of a counter 9, leading to equivalent REs of at least 10,000 symbols.

In this corpus, we see that expressions with non-trivial counters are used quite often, but often the values of the counters are also rather small. It is unclear whether this is because users do not need large counters or because there are no adequate tools for dealing with expressions involving

[2]https://github.com/bkiers/PCREParser

large counters. (On our own computers, tools such as grep complain quickly when counters grow beyond 1000.)

The majority of the expressions in the corpus have simple parse trees. Most expressions are of the form $\alpha_1 \cdots \alpha_m$, where each $\alpha_i$ is of the form $(a_1 + \cdots + a_n)^{k,\ell}$ — possibly with $n = 1$, $(k, \ell) = (1, 1)$ or $(k, \ell) = (0, \infty)$. We refer to such expressions as *CHAINs*. Therefore, *abc* is a CHAIN and so is $(A + B + C)^{3,4}(0 + 1)^*$. However, $(abc)^{1,2}$ is not, because it has a concatenation nested within the counter.[3]

We classify 2217 expressions (73.3%) from this corpus as CHAINs. Notice that the remaining expressions also contain those that use regex-specific features (e.g., lookahead) or tests beyond regular languages (e.g., backreferences), etc.

## 3.2 Snort

Snort [28] is an open source system for preventing network intrusion and, according to its web site, the most widely deployed IDS/IPS technology worldwide. It has a set of freely available *community rules*, some of which contain regular expressions. After distilling the expressions from the rules and removing duplicates, we obtained a set of 458 expressions, of which 270 (about 58.9%) use non-trivial counters. For subexpressions of the form $r^{k,\ell}$ we found values for $k$ ranging from 0 to 1024 and values for $\ell$ ranging from 1 to 1075. We found only two expressions with nesting of non-trivial counters.

Equivalent REs for freely available Snort rules therefore do not become as large as for RegExLib expressions, but Snort has a larger fraction of expressions with non-trivial counters. For example, we found 100 expressions with a three-digit number and 4 with a four-digit number for $k$. For $\ell$, 102 expressions have a three-digit value and 62 have a four-digit number. From a structural point of view, the expressions for Snort seemed to be similar to the ones from RegExLib. A quick analysis showed that at least 85.1% are CHAINs. Those that weren't CHAINs typically used a feature that we don't consider here (lookahead in regexes, for example) or a disjunction of words such as (http+ftp).

## 3.3 XML Schemas

We conducted a deep and labour-intense search for XML Schema Definitions (XSDs) on the Web. There have been studies of regular expressions in schema definitions in the past [6] but, as far as we know, this is the first study that looks at counters. We harvested XSDs from the Web by crawling the maven.org Central Repository and by using the API of Google's Custom Search Engine (CSE) [15]. We chose the Maven Central Repository because it contains high-quality source code and meta-data for many open source projects[4] and because it showed up very often when we were manually googling for XSDs. We obtained XSDs from Maven by recursively crawling its entire directory, downloading all .jar files from projects, and extracting the XSDs from those.

We used Google's CSE to find results on the entire Web. We experimented with several search engines (Bing and Yahoo) but Google returned a superset of the results we found using the alternatives and allowed us to search more precisely. We used queries of the form

```
filetype:xsd "maxoccurs=X"        (1)
```

[3]Chain regular expressions were also studied in [13, 21].
[4]See http://search.maven.org/#browse

| val | #exp | val | #exp | val | #exp | val | #exp |
|---|---|---|---|---|---|---|---|
| 2 | 2893 | 26 | 26 | 80 | 3 | 363 | 3 |
| 3 | 1609 | 27 | 5 | 85 | 3 | 365 | 4 |
| 4 | 433 | 29 | 3 | 87 | 4 | 500 | 9 |
| 5 | 3709 | 30 | 65 | 90 | 19 | 768 | 1 |
| 6 | 115 | 31 | 14 | 96 | 5 | 990 | 9 |
| 7 | 99 | 32 | 25 | 98 | 5 | 999 | 214 |
| 8 | 171 | 35 | 21 | 99 | 2020 | 1000 | 39 |
| 9 | 844 | 36 | 3 | 100 | 192 | 1024 | 1 |
| 10 | 995 | 38 | 1 | 120 | 6 | 1536 | 3 |
| 11 | 27 | 39 | 6 | 127 | 14 | 2000 | 10 |
| 12 | 79 | 40 | 16 | 128 | 23 | 3000 | 1 |
| 13 | 31 | 45 | 5 | 136 | 2 | 5000 | 16 |
| 14 | 15 | 48 | 2 | 146 | 1 | 9999 | 87 |
| 15 | 162 | 50 | 101 | 150 | 1 | 20000 | 3 |
| 16 | 83 | 51 | 2 | 176 | 1 | 65025 | 5 |
| 17 | 9 | 52 | 3 | 192 | 1 | 65535 | 25 |
| 18 | 15 | 54 | 3 | 198 | 2 | 65536 | 17 |
| 19 | 17 | 59 | 1 | 200 | 26 | 99999 | 14 |
| 20 | 303 | 60 | 20 | 250 | 2 | 200000 | 6 |
| 21 | 7 | 62 | 6 | 255 | 3 | 999999 | 24 |
| 22 | 4 | 63 | 2 | 256 | 19 | 9999999 | 2 |
| 23 | 4 | 64 | 13 | 299 | 15 | | |
| 24 | 11 | 66 | 1 | 300 | 4 | | |
| 25 | 125 | 67 | 1 | 350 | 6 | | |

(a) Non-trivial maxoccurs values in our corpus. Column "val" states the maxoccurs value; column "#exp" the number of expressions in which we found the value.

| val | #exp | val | #exp | val | #exp | val | #exp |
|---|---|---|---|---|---|---|---|
| 2 | 1099 | 9 | 3 | 19 | 1 | 54 | 3 |
| 3 | 284 | 10 | 4 | 20 | 24 | 60 | 4 |
| 4 | 73 | 11 | 9 | 22 | 1 | 85 | 3 |
| 5 | 24 | 12 | 10 | 23 | 2 | 93 | 1 |
| 6 | 11 | 13 | 1 | 24 | 3 | 127 | 1 |
| 7 | 11 | 15 | 8 | 31 | 2 | 365 | 4 |
| 8 | 9 | 16 | 8 | 51 | 1 | | |

(b) Non-trivial minoccurs values in our corpus. Column "val" states the minoccurs value; column "#exp" the number of expressions in which we found the value.

Figure 1: Non-trivial counter values occurring in our corpus.

(and variations thereof) to explicitly search for schemas with a maxoccurs value of $X$ and

```
filetype:xsd "maxoccurs=X..Y"        (2)
```

to search for schemas with a maxoccurs value $Z$ such that $X \leq Z \leq Y$. (Similarly for minoccurs.) Since Google CSE only allows to process around 100 queries per day, it was infeasible within our time constraints to conduct a query of type (1) for each possible counter value. Furthermore, a single query to Google's CSE only returns at most 100 links. We therefore used queries of type (2) mainly for discovering new values of $X$ that return non-empty results and then use type (1) to find all results with this new value.

**Data Cleaning.** We extracted 4808 XSDs from the Maven Central Repository. Among them, 285 (about 6%) use a non-trivial counter value. Additionally, through Google we found 12,211 unique URLs outside Maven Central that match at least one of the queries of type (1) or (2) above. About 1400 of these URLs did not directly point to an XSD but to an HTML file that, somewhere, contains a link or path of

links to an XSD. By resolving these links recursively and by also downloading XSDs that are referenced in other XSDs, we found 8944 more URLs claiming to point to a file with .xsd extension. From this data set, we removed everything that is not a valid XSD. This operation resulted in a total of 3259 unique URLs containing XSDs with non-trivial counter values that we obtained through Google.

We then removed duplicate files. More precisely, we parsed all XSDs we found, normalized the whitespace, and removed files that occurred more than once after this operation. In total, we obtained 1191 unique, well formed XSD files that contain non-trivial counter values; 906 through Google and 285 from the Maven Central Repository. This set of XSDs forms the input of the study that follows.

**Description of the Corpus.** The 1191 schemas from our repository contain 9389 regular expressions with non-trivial counter values. Our repository (raw data and the clean corpus) can be found at http://regx.github.io/.

The distribution of counter values in our corpus is summarized in Fig. 1. Here we present, per possible finite maxoccurs value (Fig. 1a) and minoccurs value (Fig. 1b), the number of regular expressions in which it was found. The largest value we found was 9,999,999 and occurred in two different versions of an XSD from IBM, related to electronic data interchange for administration, commerce, and transport (EDIFACT).[5]

Notice that the total sum of count values in Fig. 1 is larger than 9389, because a single expression can contain multiple occurrences of counters. One example is the expression $(ab^{2,12})^{0,65535}$, which we found in a schema that seems to be related to the MPEG-7 standard. A brief inspection of our corpus exposes that several major companies use regular expressions with non-trivial counters, for example: the Amazon Simple Storage Service (S3) schema[6] (maxoccurs 100), FedEx's web services[7] (maxoccurs 12, 99, 999), Paypal's WSDL interface[8] (maxoccurs 100, 1000), and several schemas for the MPEG-7 standard[9] use maxoccurs values up to 65636. We note that our corpus by no means contains all available XSDs on the Web with non-trivial counter values. Some prominent examples that we did not find automatically can be found in Microsoft's MSDN library, which contains schemas with maxoccurs values of, e.g., 50, 100, 256, 1000, and 100,000.[10]

**Structural Analysis.** We analyzed the structure of regular expressions in our corpus, in the same spirit as was done by Bex et al. [6]. This allows us to compare results with [6], which was performed a decade ago. Bex et al. observed that a very large percentage of expressions in practice only use each alphabet symbol at most once. This observation still holds: in our corpus, 98% of all regular expressions use alphabet symbols at most once. A little bit of care is needed when comparing this number to [6]: In the latter study, non-trivial counter values in expressions were first rewritten to trivial ones, e.g., $b^{4,5}$ was rewritten to *bbbbb*?. As such, the expression $b^{4,5}$ would not be counted as having a single $b$. The highest numbers of occurrences for the same symbol in one expression we found was 200.

| depth | star | counter | iteration |
|---|---|---|---|
| 0 | 7333 | | |
| 1 | 2036 | 9361 | 9313 |
| 2 | 19 | 27 | 73 |
| 3 | 1 | 1 | 3 |

Figure 2: Star-, counter-, and iteration depth for expressions in our corpus.

When looking at the structural complexity of the expressions in our corpus, we see that they have shallow parse trees. For computing parse tree depth, we first normalize expressions by exploiting associativity of disjunction and concatenation. That is, when we see an expression of the form $((a+b)+c)+d$, we normalize it to $a+b+c+d$ and say that its parse tree has depth two, rather than four. After this preprocessing step, 8132 expressions have depth 3; 620 have depth 4; 542 have depth 5; 36 have depth 6; and 59 have depth ranging from 7 to 9.

Next, we looked at the nesting of repetition operators. Fig. 2 contains the *star depth* of the expressions in our corpus, i.e., the number of nestings of operators that allow an unbounded number of repetitions of their associated subexpression, i.e., operators of the form $^{k,\infty}$ for some $k \in \mathbb{N}$. (So, both expressions $(a^{0,\infty})^{2,100}$ and $a^{2,\infty}$ have star depth one.) Not surprisingly, most expressions do not use any nesting of such expressions at all. Only nine expressions in our entire corpus use nesting depth 2 or 3.

Also contained in Fig. 2 is the *counter depth* of regular expressions, which is the nesting depth of counters of the form $(k, \ell)$ where at least one of $k$ or $\ell$ is a natural number larger than 1. (So, both expressions $(a^{0,\infty})^{2,100}$ and $a^{2,\infty}$ have counter depth one.)

Finally, we define the *iteration depth* to be the nesting depth of any form of iteration, that is stars and counters. The former of the two aforementioned expressions has iteration depth two and the latter has one. Fig. 2 suggests that expressions with non-trivial counters in XSDs are structurally rather simple. We suspect that this is because highly complex regular expression in an XSD may indicate that the design of the underlying XML database is not very elegant. A closer look at the expressions showed that 8330 expressions from our corpus with non-trivial counter values are CHAINs, which is about 88.7%. (Within the pool of all regular expressions we found in schemas, about 86% were CHAINs.)

## 4. MEMORY-EFFICIENT AUTOMATA

Most algorithms that process regular expressions with counting (notably, grep and all XML Schema validation tools that we are aware of) first convert them to ordinary regular expressions by expanding the counters. Subsequently, these (already exponentially large) expressions are converted to finite automata. This exponential blow-up can be avoided by directly converting the expressions into a much more

[5]https://github.com/DFDLSchemas/EDIFACT/raw/master/EDIFACT-SupplyChain-D03B/EDIFACT-SupplyChain-Messages-D.03B.xsd. This schema contains other large values too.

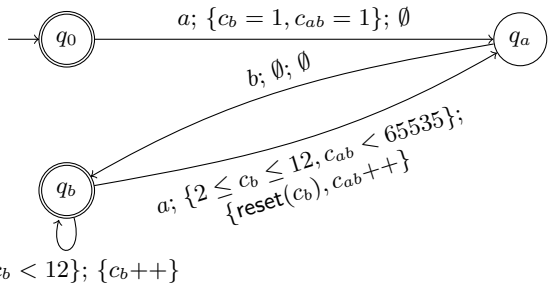
[6]http://doc.s3.amazonaws.com/2006-03-01/AmazonS3.xsd

[7]http://www.fedex.com/templates/components/apps/wpor/secure/downloads/xml/Aug09/Advanced/ShipService_v7.xsd

[8]http://www.paypalobjects.com/wsdl/eBLBaseComponents.xsd

[9]http://standards.iso.org/ittf/PubliclyAvailableStandards/MPEG-7_schema_files/mpeg7-v2.xsd

[10]http://msdn.microsoft.com/en-us/library/cc233001.aspx, http://msdn.microsoft.com/en-us/library/jj583348.aspx, http://msdn.microsoft.com/en-us/library/gg309601.aspx




Acceptance condition: $c_{ab} \leq 65535$.

Figure 3: A Counter Automaton for $(ab^{2,12})^{0,65535}$.

memory-efficient representation: finite automata with counters. Such automata have been used and implemented before, for example in the context of deep packet inspection in networks [27], where significant improvements in speed and memory consumption were reported.

We provide a gentle introduction to such automata here and we compare them to other evaluation algorithms in Section 6.1. The performance gain is clear: compared to the above mentioned naive method for evaluation, we see improvements of several orders of magnitude. The differences for incremental evaluation become even more drastic.

**Counter Automata by Example.** *Counter automata* are a natural representation of expressions with counters that is easier to deal with in algorithms. We first discuss a counter automaton by example and explain them more formally next. Figure 3 depicts a counter automaton for the expression $(ab^{2,12})^{0,65535}$, which we found in our practical study in a schema related to the MPEG-7 standard. An ordinary non-deterministic finite state automaton for this expression would require 851955 states. The counter automaton only has three: a start state $q_0$ and states for each occurrence of a symbol in the expression – $q_a$ for $a$ and $q_b$ for $b$. (Given a regular expression with counters that has $n$ occurrences of symbols, we can always construct an equivalent counter automaton with $n + 1$ states.)

The automaton has two *counters*: $c_b$, for counting the number of $b$'s and $c_{ab}$ for counting how many times it has seen a word that matches $ab^{2,12}$. Initially, all counters are set to 1.

The state transitions in the automaton consist of three ingredients: a *symbol*, a set of *guards*, and a set of *counter updates*. For example, the transition from $q_0$ to $q_a$ reads symbol $a$, has guards $c_b = 1$ and $c_{ab} = 1$, and an empty set of counter updates. So, the transition reads an $a$-symbol and can be made when $c_b = c_{ab} = 1$. The next interesting transition is the $b$-loop on $q_b$. It allows to read a $b$ when $c_b < 12$, and it increases $c_b$ by one. Finally, the transition from $q_b$ to $q_a$ reads an $a$, can be performed if $c_b$ is between 2 and 12 and $c_{ab} < 65535$. It increases $c_{ab}$ by one and overwrites $c_b$ with 1.

The automaton accepts if it is in an accepting state (double circles) and if the *acceptance condition* holds, that is, $c_{ab}$ is at most 65535. It is easy to see that the automaton accepts precisely the words that match $(ab^{2,12})^{0,65535}$.

**Formal Definition.** We now define counter automata formally. Their definition is rather technical but essentially follow the lines of the example above. Readers who already understand the main idea can safely skip this definition.

Formally, counter automata are an extension of non-deterministic finite automata, which we briefly recall here for the sake of clarifying notation. A *non-deterministic finite automaton (NFA)* is a tuple $N = (Q, \Sigma, \delta, I, F)$, where $Q$ denotes its set of states, $\Sigma$ its alphabet, $I$ its set of initial states, and $F$ its set of final or accepting states. The transition rules $\delta$ are of the form $q_1 \xrightarrow{a} q_2$, indicating that reading an $a \in \Sigma$ in state $q_1$ can bring the automaton to state $q_2$. Acceptance is defined in the standard manner. We denote by $L(N)$ the set of words accepted by $N$.

Counter automata extend NFAs with *counter variables* or *counters*. We follow [11] in their definition. Let $C$ be a finite set of *counter variables* and $\alpha : C \to \mathbb{N}$ be a function assigning a value to each counter variable. A *guard over* $C$ is a function $\phi : C \to (\mathbb{N} \times \mathbb{N}_\infty)$, where $\mathbb{N}_\infty$ denotes $\mathbb{N} \cup \{\infty\}$. Furthermore, we will require that, if $\phi(c) = (k, \ell)$, then $k \leq \ell$. The semantics of a guard is defined as follows. We say that a counter assignment $\alpha$ *satisfies* guard $\phi$ if, for every $c \in C$, whenever $\phi(c) = (k, \ell)$, then $k \leq \alpha(c) \leq \ell$. Intuitively, this means that a guard defines, for each counter $c$, a minimum allowed value $k$ and a maximum allowed value $\ell$. As such, guards are used in counter automata to model the upper and lower bounds of counters in regular expressions. By $\alpha \models \phi$ we denote that the counter assignment $\alpha$ satisfies guard $\phi$. By Guards($C$) we denote the set of guards over $C$.

A *basic update over* $C$ is a partial function $\pi : C \to \{\mathsf{reset}, \mathsf{increment}\}$. When $\pi(c) = \mathsf{reset}$, this means that $c$ should be reset to one and when $\pi(c) = \mathsf{increment}$ it means that $c$ should be incremented by one. (If $\pi$ is undefined on $c$, we leave $c$ unchanged.) By Basic-Up($C$) we denote the set of all basic updates over $C$.

Definition 4.1. *A (non-deterministic)* automaton with counters *(NFA$^\#$) is a 6-tuple $A = (Q, q_0, C, \delta, F, \tau)$ where*

- *$Q$ is the finite set of states;*
- *$q_0 \in Q$ is the initial state;*
- *$C$ is the finite set of counter variables;*
- *$\delta \subseteq Q \times \Sigma \times$ Guards($C$) $\times$ Basic-Up($C$) $\times Q$ is the transition relation;*
- *$F : Q \to$ Guards($C$) is the acceptance function; and*
- *$\tau : C \to \mathbb{N}$ assigns a maximum value to every counter variable.*

Intuitively, an NFA$^\#$ $A$ can make a transition $(q, a, \phi, \pi, q') \in \delta$ whenever it is in state $q$, reads label $a$, and the guard $\phi$ is satisfied by the current values of the counter variables. It then updates the counter variables according to the update $\pi$ (in a way which we explain next) and moves into state $q'$. To explain the update mechanism formally, we use *configurations*. A *configuration* is a pair $(q, \alpha)$ where $q \in Q$ is the current state and $\alpha : C \to \mathbb{N}$ is the function mapping counter variables to their current value. An update $\pi$ transforms $\alpha$ into $\alpha'$ by setting $\alpha'(c) := 1$ whenever $\pi(c) = \mathsf{reset}$ and $\alpha'(c) := \alpha(c) + 1$ whenever $\pi(c) = \mathsf{increment}$. We sometimes abuse notation and denote $\alpha'$ by $\pi(\alpha)$.

Let $\alpha_{\text{init}}$ be the function mapping every counter variable to 1. The *initial configuration* $\gamma_0$ is $(q_0, \alpha_{\text{init}})$. A configuration $(q, \alpha)$ is *final* if $\alpha \models F(q)$. A configuration $\gamma' = (q', \alpha')$ *immediately follows* a configuration $\gamma = (q, \alpha)$ by reading $a \in \Sigma$, denoted $\gamma \to_a \gamma'$, if there exists $(q, a, \phi, \pi, q') \in \delta$ with $\alpha \models \phi$ and $\alpha' = \pi(\alpha)$.

For a word $w = a_1 \cdots a_n$ and two configurations $\gamma$ and $\gamma'$, we denote by $\gamma \Rightarrow_w \gamma'$ that $\gamma \to_{a_1} \cdots \to_{a_n} \gamma'$. A

configuration $\gamma$ is *reachable* if there exists a word $w$ such that $\gamma_0 \Rightarrow_w \gamma$. A word $w$ is *accepted* by $A$ if $\gamma_0 \Rightarrow_w \gamma_f$ where $\gamma_f$ is a final configuration. We denote by $L(A)$ the set of words accepted by $A$.

The *size* of a transition $\theta$ or acceptance condition $F(q)$ is the total number of occurrences of alphabet symbols, states, counter variables, and Boolean connectives which occur in it, plus the size of the binary representation of each integer occcurring in it. In the same spirit, the size of $A$, denoted by $|A|$, is $|Q| + \sum_{q\in Q} \log \tau(q) + |F(q)| + \sum_{\theta\in\delta} |\theta|$.

It is known that RE$^\#$ expressions can be efficiently translated into equivalent NFA$^\#$s by applying a natural extension of the known Glushkov construction [11, 29].

# 5. INCREMENTAL EVALUATION

We present a new incremental evaluation algorithm for regular expressions with counters. To the best of our knowledge, this is the first such algorithm that avoids an exponential translation of the expressions. In Section 6 we see that incremental evaluation outperforms evaluation from scratch by factors up to three orders of magnitude and, shifting to automata with counters improves up to another three orders of magnitude. In fact, such speed-ups can be expected: incremental evaluation is an exponential improvement over evaluation from scratch; and doing it with counter automata is an yet another exponential improvement over doing it with ordinary automata.

As a warm-up, we recall how to incrementally evaluate ordinary regular expressions on words. The technique was first described by Patnaik and Immerman [25] and was studied in more detail and implemented by Balmin et al. [2].

**Incremental Evaluation of REs and NFAs.** Assume that we have a word $w = a_1 \cdots a_n \in \Sigma^*$ and a regular expression $r$ and we want to incrementally maintain whether $w \in L(r)$. The incremental evaluation problem consists of two phases: a one-time *preprocessing phase* in which we construct an auxiliary data structure that we will maintain during updates; and an *evaluation phase* in which updates of the form $\mathsf{relabel}(i, a)$, $\mathsf{delete}(i)$, or $\mathsf{insert}(i, a)$ arrive, where $i \in \{1, \ldots, n\}$ is a position in the word and $a \in \Sigma$ is a symbol. The updates do the obvious: $\mathsf{relabel}$ changes $a_i$ to $a$, $\mathsf{delete}$ deletes the symbol $a_i$ and results in a word of length $n-1$, and $\mathsf{insert}(i, a)$ inserts an $a$ before position $i$, resulting in a word of length $n + 1$. We denote the newly obtained word by $w'$. In the evaluation phase, the task is to be able to say whether $w' \in L(N)$ quickly after the update arrives. So, after a one-time preprocessing phase, we want to be able to deal with (multiple) updates quickly. Here, we describe a method for incremental evaluation that uses $\mathcal{O}(n \cdot |r|^2)$ time for preprocessing and then, in the update phase, can answer in time $\mathcal{O}(|r|^3 \cdot \log n)$ per update whether the new word $w'$ is still in $L(r)$ or not. Therefore, when $n$ is large, the procedure is much faster than re-evaluating the expression from scratch, which would require at least linear time in $n$.

The incremental update algorithm works on the NFA for $r$, which can be constructed in linear time and which we denote by $N = (Q, \Sigma, \delta, I, F)$. We first describe the auxiliary data structure we will maintain during the updates. For each $i, j$, $1 \le i < j \le n$, let $T_{ij}$ be the transition relation $\{(p, q) \mid p, q \in Q, p \xrightarrow{a_i \cdots a_j} q\}$, where $p \xrightarrow{a_i \cdots a_j} q$ denotes that $N$ can reach state $q$ when it starts in state $p$ and reads $a_i \cdots a_j$. Note that $T_{ij} = T_{ik} \bowtie T_{(k+1)j}$, (with $i < k < j$), where $\bowtie$ denotes the natural join on binary relations, that is, $T_{ij}$ contains all $(x, z)$ such that there is a $y$ with $(x, y) \in T_{ik}$ and $(y, z) \in T_{(k+1)j}$.

For simplicity, assume first that $n$ is a power of 2, say $n = 2^k$. The main idea is to keep as auxiliary information just the $T_{ij}$ for intervals $[i, j]$ obtained by recursively splitting $[1, n]$ into halves, until $i = j$. More precisely, consider the transition relation tree $\mathcal{T}_n$ whose nodes are sets $T_{ij}$, defined inductively as follows:

- the root is $T_{1n}$;
- each node $T_{ij}$ for which $j - i > 0$ has children $T_{ik}$ and $T_{(k+1)j}$ where $k = i - 1 + \frac{j-i+1}{2}$; and
- the $T_{ii}$ are the leaves, for all $1 \le i \le n$.

Note that $\mathcal{T}_n$ has $n + (n/2) + \cdots + 2 + 1 = 2n - 1$ nodes and has depth $\log n$. Thus, the size of the auxiliary structure is $\mathcal{O}(n \cdot |Q|^2)$.

The *preprocessing phase* consists of building this auxiliary data structure. Notice that, once we have $\mathcal{T}_n$, it is easy to decide whether $w \in L(N)$. Indeed, $w \in L(N)$ if and only if $(q, f) \in T_{1n}$ for some $q \in I$ and $f \in F$. Therefore, we only have to show that this auxiliary data structure can be updated efficiently.

We now describe the *evaluation phase*. For simplicity, consider the case when one update occurs, changing the label of the symbol in $w$ at position $k$ to $b$. That is, the new word is $w' = a_1 \cdots a_{k-1} b a_{k+1} \cdots a_n$. The relations $T_{ij} \in \mathcal{T}_n$ that are affected by the updates are those lying on the path from the leaf $T_{kk}$ to the root of $\mathcal{T}_n$. Denote this set of relations by $I$ and notice that it contains at most $\log n$ relations. The tree $\mathcal{T}_n$ is updated by recomputing the relations in $I$ bottom-up as follows: First, the leaf relation $T_{kk}$ is set according to $\delta$ and $b$. Then each $T_{ij} \in I$ with children $T'$ and $T''$, of which one has been recomputed, is replaced by $T' \bowtie T''$. Thus, at most $\log n$ relations have been recomputed, each in time $\mathcal{O}(|Q|^3)$, yielding a total time of $\mathcal{O}(|Q|^3 \cdot \log n)$.[11] If we arrive at the root, we know that $w' \in L(N)$ if and only if $(q, f) \in T_{1n}$ for some $q \in I$ and $f \in F$.

The above approach can easily be adapted to words whose length is not a power of 2. Further, the auxiliary data structure has size $\mathcal{O}(n \cdot |Q|^2)$. Finally, handling updates in which elements are inserted or deleted is also done in [2], but then some precautions have to be taken in order to make sure that the tree $\mathcal{T}_n$ remains properly balanced.

THEOREM 5.1 ([25]). *Incremental evaluation of an RE $r$ on a word $w$ is possible with preprocessing time $\mathcal{O}(n \cdot |r|^3)$, an auxiliary data structure of size $\mathcal{O}(n \cdot |r|^2)$, and time $\mathcal{O}(\log n \cdot |r|^3)$ per update.*

**The Incremental Evaluation Algorithm for RE$^\#$s.** The incremental evaluation algorithm for RE$^\#$s extends the one for REs. It is based on NFA$^\#$s rather than NFAs and it does not translate RE$^\#$s to exponentially larger automata.

Intuitively, the algorithm for RE$^\#$s follows the same lines as the algoritm for REs, but it (1) stores different information at the nodes of the auxiliary tree and, subsequently also (2) uses a different algorithm for joining the information in neighboring nodes. We now describe these two changes.

*New information in the auxiliary tree.* In the algorithm for REs, a tuple $(p, q)$ is in $T_{ij}$ if and only if the automaton

[11] Using fast matrix multiplication algorithms, this time can be improved to $\mathcal{O}(|Q|^\omega \cdot \log n)$ (where $\omega$ denotes the best known bound for matrix multiplication).

$N$ can read $a_i \cdots a_j$ when going from state $p$ to state $q$. When we want to do something similar for NFA$^\#$s, we need to take the counters in account. To this end, a *(general) update over* $C$ is a function $\pi : C \to (\{\mathsf{inc}, \mathsf{assign}\} \times \mathbb{N})$. For readability we often consider $\pi$ as a set of statements of the form $\mathsf{inc}(c,k)$ or $\mathsf{assign}(c,k)$. (Hence, $\pi$ only contains one of $\mathsf{inc}(c,k)$ or $\mathsf{assign}(c,k)$ for each counter $c$.) Intuitively, when $\pi$ contains $\mathsf{inc}(c,k)$, then counter $c$ should be incremented with $k$. If it contains $\mathsf{assign}(c,k)$, then $c$ is assigned the value $k$, so it is overwritten. We need $\mathsf{assign}$ because if we perform some sequence of transitions in which a counter $c$ is reset, then $c$ should be assigned a value $k$, which will be the number of increments that were done to $c$ *after the last reset* plus 1.[12] To this end, we use a rule $\mathsf{assign}(c,k)$. If we model a sequence of updates without a reset, then we use a rule $\mathsf{inc}(c,k)$, where $k$ is the number of increments to $c$ in the sequence. We write Updates($C$) for the set of all general updates over $C$.

Formally, an update $\pi \in \text{Updates}(C)$ transforms counter assignment $\alpha$ into $\alpha'$ by setting $\alpha'(c) = k$ if $\pi$ contains $\mathsf{assign}(c,k)$ and setting $\alpha'(c) = \alpha(c)+k$ if $\pi$ contains $\mathsf{inc}(c,k)$. We sometimes denote $\alpha'$ by $\pi(\alpha)$. Thus, if $\alpha$ is transformed into $\alpha'$ by $\pi$, then $\pi(\alpha)(c) = \alpha'(c)$. For the remaining definitions of this section, we fix a NFA$^\#$ $A = (Q, q_0, C, \delta, F, \tau)$.

In the incremental evaluation algorithm for RE$^\#$s, the sets $T_{ij}$ contain *transformation tuples*:

Definition 5.2. A *transformation tuple* of $A$ is a quadruple $t \in Q \times \text{Guards}(C) \times \text{Updates}(C) \times Q$. Tuple $t = (p, \phi, \pi, q)$ is *consistent* with a string $w$ if, for every configuration $\gamma = (p, \alpha)$ such that $\alpha \models \phi$, there is a configuration $\gamma' = (q, \alpha')$ such that $\gamma \Rightarrow_w \gamma'$ in $A$ and $\alpha' = \pi(\alpha)$.

That is, a transformation tuple is consistent with $w$ if it captures the effect of $w$ on the NFA$^\#$, i.e., if it describes how configurations that match it change by reading $w$.

Example 5.3. Consider the NFA$^\#$ $(Q, q_0, C, \delta, F, \tau)$ from Figure 3. The tuple $t_1 = (q_0, \phi_1, \pi_1, q_b)$ where $\phi_1$ expresses

$$c_b = 1 \quad \text{and} \quad c_{ab} = 1$$

and $\pi_1$ contains $\mathsf{inc}(c_b, 2)$ and $\mathsf{inc}(c_{ab}, 0)$ is consistent with $w = abbb$. (Intuitively, if the NFA$^\#$ starts in $q_0$ and reads $w$, it ends up in state $q_b$ and increases $c_b$ by two.) If $\phi_1$ would only require $1 \le c_b \le 2$, the tuple would *not* be consistent with $w$, because we can only read $a$ in $q_0$ if $c_b = 1$.

The tuple $t_2 = (q_b, \phi_2, \pi_2, q_b)$ where $\phi_2$ expresses

$$1 \le c_b \le 9 \quad \text{and} \quad 1 \le c_{ab} \le 65534$$

and $\pi_2$ contains $\mathsf{assign}(c_b, 3)$ and $\mathsf{inc}(c_{ab}, 1)$ is consistent with *babbb* and also with *bbbabbb*. It is not consistent with *abbb* (because we cannot go from $q_b$ to $q_a$ if $c_b = 1$) or with *bbbbabbb* (because we cannot read $b$ in $q_b$ if $c_b = 12$). □

So, the auxiliary data structure for incremental evaluation of RE$^\#$s is a binary tree $\mathcal{T}_n$ whose nodes are sets $T_{ij}$ that contain precisely the transformation tuples that are consistent with $a_i \cdots a_j$.

*Joining of transformation tuples.* It remains to explain how the transformation tuples can be computed and updated. To this end, we merely have to redefine the *join* operation $\bowtie$ we used in the algorithm for REs. In other words, we must define when transformation tuples $t_1 = (p_1, \phi_1, \pi_1, q_1)$ and $t_2 = (p_2, \phi_2, \pi_2, q_2)$ can be joined. To this end, we say that $t_1$ is *compatible with* $t_2$ if $q_1 = p_2$ and there is a counter assignment $\alpha$ such that $\alpha \models \phi_1$ and $\pi_1(\alpha) \models \phi_2$. Intuitively, if $t_1$ is consistent with word $w_1$, $t_2$ is consistent with word $w_2$, and $t_1$ is compatible with $t_2$, then there is a counter assignment $\alpha$ such that $(p_1, \alpha) \Rightarrow_{w_1 w_2} (q_2, \alpha')$ with $\alpha' = \pi_2(\pi_1(\alpha))$.

Example 5.4. For the NFA$^\#$ of Figure 3, the tuple $t_1$ from Example 5.3 is compatible with $t_2$. For example, for the assignment $\alpha_0$ with $c_b = 1$ and $c_{ab} = 1$ we have that $\alpha_1 := \pi_1(\alpha_0)$ maps $c_b$ to 3 and $c_{ab}$ to 1. Therefore, $\alpha_1 \models \phi_2$. As such, when in configuration $(q_0, \alpha_0)$ the NFA$^\#$ can read strings *abbbbabbb* or *abbbbbbabbb*.

We now define how transformation tuples are joined.

Definition 5.5. Let $t_1 = (p, \phi_1, \pi_1, r)$ be compatible to $t_2 = (r, \phi_2, \pi_2, q)$. Then the *join of* $t_1$ *with* $t_2$, denoted $t_1 \bowtie_c t_2$, is the tuple $(p, \phi, \pi, q)$ such that, for every counter assignment $\alpha$, we have *(i)* $\alpha \models \phi \Leftrightarrow (\alpha \models \phi_1 \wedge \pi_1(\alpha) \models \phi_2)$, and *(ii)* $\pi(\alpha) = \pi_2(\pi_1(\alpha))$.

Example 5.6. The join of $t_1$ with $t_2$ from Example 5.3 is $(q_0, \phi, \pi, q_b)$, where $\phi$ expresses

$$c_b = 1 \quad \text{and} \quad c_{ab} = 1$$

and $\pi$ contains $\mathsf{assign}(c_b, 3)$ and $\mathsf{inc}(c_{ab}, 1)$. Consider tuple $t_1' = (q_b, \phi_1', \pi_1', q_b)$ where $\phi_1'$ expresses

$$3 \le c_b \le 9 \quad \text{and} \quad 1 \le c_{ab} \le 65535$$

and $\pi_1'$ contains $\mathsf{inc}(c_b, 2)$ and $\mathsf{inc}(c_{ab}, 0)$, then $t_1'$ is compatible with $t_2$ and $t_1' \bowtie_c t_2$ is $(q_b, \phi', \pi', q_b)$, where $\phi'$ expresses

$$3 \le c_b \le 7 \quad \text{and} \quad 1 \le c_{ab} \le 65534$$

and $\pi'$ contains $\mathsf{assign}(c_b, 3)$ and $\mathsf{inc}(c_{ab}, 1)$. Intuitively, we have $3 \le c_b \le 7$ in the result because $\phi_1'$ requires $3 \le c_b$ and $\phi_2$ requires $\pi_1'(c_b) \le 9$. □

When regular expressions contain few alphabet symbols and large counter values, as is typically the case in the CHAINs we found in the practical study (Section 3), the new algorithm performs much better. This is seen most clearly when considering RE$^\#$s of the form $r = a^{k,k}$. Assuming unit cost for basic arithmetic on numbers up to $k$, incremental evaluation for their NFA$^\#$s on a word of length $n$ costs preprocessing time $\mathcal{O}(n)$ and update time $\mathcal{O}(\log n)$, whereas the algorithm of Section 5 would cost preprocessing time $\mathcal{O}(n \cdot 2^{3|r|})$ and update time $\mathcal{O}(2^{3|r|} \cdot \log n)$. The large speed-ups we obtain in Section 6 are therefore due to the fact that expressions with large counters we found in practice seem to have a rather simple structure, which exploits the potential of NFA$^\#$s close to optimal.

# 6. EXPERIMENTS

We performed an extensive experimental study on regular expression evaluation algorithms; incremental or otherwise. We implemented all algorithms and benchmarks in Java, mainly because it is very portable and wide-spread. To ensure comparability between measurements, we also reimplemented all other algorithms in Java (and on the same underlying data structures). Since Java uses garbage collection, there are several caveats (which we'll discuss later) for measuring memory consumption as well as execution time. All our experiments are conducted on a machine with an Intel Core i7-2600K. We allocated a heap of at least 3 GiB for

[12] The plus one term comes from the fact that the counters are reset to 1.

all JVM instances during our tests. All tests were executed on a 64-bit JVM.

We perform two kinds of experiments: *evaluation from scratch* (i.e., not dealing with updates) and *incremental evaluation* (dealing with updates). In both settings, we compare algorithms based on NFAs to algorithms based on NFA$^\#$s. The NFA$^\#$ variants are several orders of magnitude faster than the algorithms based on NFAs, scale much better, and consume much less memory.

## 6.1 Evaluation from Scratch

The experimental task in this section is the following:

| | | |
|---|---|---|
| PROBLEM | : | Evaluation |
| INPUT | : | A regular expression with counting $r$ and a word $w$. |
| QUESTION | : | Is $w \in L(r)$? |

We compare the following algorithms:

**(RE#):** Fast squaring algoritm for RE$^\#$
**(NFAsim):** Building and simulating an NFA
**(NFA$^\#$sim):** Building and simulating a NFA$^\#$
**(NFApre):** Preprocessing for incremental NFA evaluation
**(NFA$^\#$pre):** Preprocessing for incremental NFA$^\#$ eval

We discuss the five algorithms next. The benchmark (RE#) is an optimization of an algorithm by Kilpeläinen and Tuhkanen [18] which is the only polynomial time algorithm for evaluating RE$^\#$ we are aware of that does not translate to automata.[13] It considers the parse tree of the expression $r$ and computes bottom-up, for each node $u$ in the parse tree, all pairs $(i, j)$ of positions in $w = a_1 \cdots a_n$ such that the subexpression rooted at $u$ matches the subword $a_i \cdots a_j$. For example, when node $u$ is labeled by a disjunction, its corresponding relation can be obtained by taking the union of the relations of $u$'s children. If $u$ is a concatenation, its relation is the composition (or natural join) of the relation of its children. If $u$ is a $*$, we need to take the transitive-reflexive closure of the relation of its child. Lastly, if $u$ is a counting operator (say, a $k$-fold iteration), its relation is the $k$-fold composition of the relation of its child. In this latter case lies the optimization. Instead of performing $O(k)$ compositions as in [18], we comput the $k$-fold composition by fast squaring, which only costs $O(\log k)$ compositions. This algorithm is known to have worst-case complexity $O(|w|^m \cdot |r|)$, where $m$ is the best bound for multiplying two $|w| \times |w|$ zero-one matrices. We refer the reader to [18, 20] for further details of the algorithms and their complexity.

The (NFAsim) algorithm translates $r$ into an NFA, which is then evaluated on $w$. The bottleneck of this approach lies at dealing with the counters of $r$. A counter value of $k$ typically results in $\Omega(k)$ states in the NFA. Since the counter value is represented by $\log k$ bits in $r$, this constitutes exponential cost in $|r|$.

In (NFA$^\#$sim), we avoid this blow-up by compiling $r$ into a NFA$^\#$ of size only $O(|r|)$, which we then evaluate on $w$.

The algorithms (NFApre) and (NFA$^\#$pre) are the preprocessing phases of the incremental evaluation algorithms of Section 5. Once these preprocessing phases are finished, one can decide in constant time whether $w \in L(r)$ by inspecting the root of the auxiliary data structure. By comparing the cost of the (—pre) with the (—sim) variants we can therefore gauge the cost of the one-time overhead for building an auxiliary data structure that allows incremental evaluation for future updates.

We present three benchmarks: a *sanity check*, a *worst-case synthetic* benchmark with much non-determinism in expressions, and a benchmark that shows how *large counter values* behave in real-world and synthetic expressions. The benchmarks use expressions that are structurally very simple, because (1) simplicity in the setup allows for a better understanding of the results and (2) we did not see different results on more complex setups; all behaviour we saw in more complex (real-world or synthetic) settings could be explained by these three experiments.

In order to get reliable measurements in Java we repeated each experiment about five hundred times, discarded the best and worst 10% to get rid of garbage cleaning artefacts, and took the average of the remaining ones. (We observed that this way we consistently obtained the same measurements when experiments were repeated.) Since this procedure makes experiments lengthy, we only measured up to 30 seconds. The charts we present are optimized for readability and drastically summarise the data points we measured.

**Sanity Check Benchmark.** First we compared the five candidates for $r = a^*$ and $w$ being words containing only $a$'s (Fig. 4, left). The rationale behind this test is to see how the five methods compare *if the NFA$^\#$ has no advantage* over the other methods and to *gauge the overhead of NFA$^\#$s versus NFAs* in our implementation. Since $a^*$ can be converted to an NFA or NFA$^\#$ within microseconds, we only see the time required to process the word. Here, fast squaring (RE#) is by far the slowest of all. The reason is its high complexity in terms of $|w|$. The incremental variants are about an order of magnitude slower than the non-incremental variants. This factor can be explained by the extra cost of building the external data structure for the incremental algorithms. In this setting where the NFA$^\#$-based algorithms do not have any advantage over the NFA-based algorithms, the NFA$^\#$-based algorithms are marginally slower.

**Real-world and large counters.** Next we compared the candidates on real-world and synthetic expressions that use very large counter values. The underlying idea of this test is to see how the candidates compare if the NFA$^\#$s can exploit counters to their maximum advantage. In this experiment we noticed that the behavior for real-world expressions was the exactly same than on synthetic ones. In Fig. 4, middle, we present for increasing values of $n$, the expression $r = a^{(n-n/4),(n+n/4)}$, so the range of allowed lengths is about $n/2$. We then perform membership tests of words of length $n$, $n-n/4$, $n-n/4-1$, $n+n/4$, and $n+n/4+1$ and take the average of the evaluation times we measured. It is striking that, throughout this experiment, the preprocessing phase for incremental NFA$^\#$ evaluation is faster than conversion to an NFA and evaluating it.

**Synthetic worst-case.** Finally we investigated synthetic expressions with counting, but with a high amount of non-determinism (Fig. 4, right). We consider expressions $r$ for which the minimal deterministic finite automaton is *double* exponentially larger than $r$. We perform this test because the two previous experiments concerned very simple deterministic expressions and it is well-known that non-determinism complicates fast evaluation.

[13] Notice that rewriting an RE$^\#$ into an equivalent RE already costs exponential time.

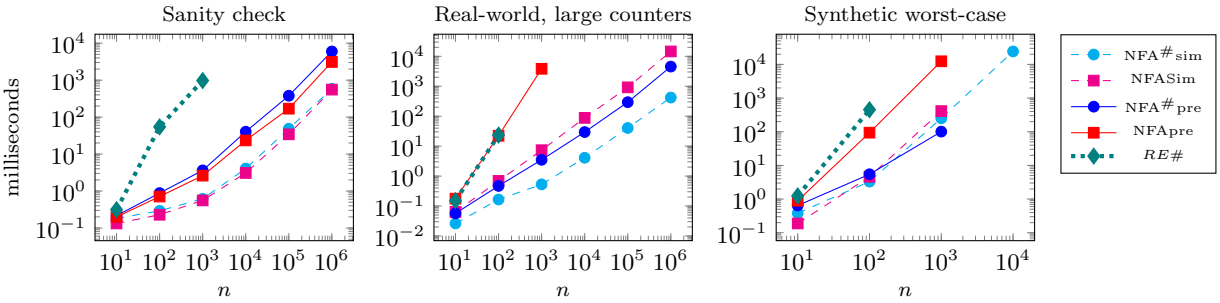



Figure 4: Evaluation from scratch (log-log scale).

The most striking and perhaps counter-intuitive fact about this experiment is that (NFA$^{\#}$pre) is the fastest evaluation algorithm in this case. We believe that this is so because the evaluation algorithms for NFAs (resp., NFA$^{\#}$s) need to store and maintain a large number of possible states (resp., configurations) while reading the word. Since (NFA$^{\#}$pre) works completely differently and can summarize configurations, it even outperforms the standard evaluation algorithms.

A second striking fact is that non-determinism combined with large counter values can bring most evaluation algorithms to their knees quickly. This may be the reason why the XML Schema specification only allows very limited non-determinism in regular expressions (cfr. [11]).

## 6.2 Incremental evaluation

We compare the two variants of incremental evaluation:

**(incNFA)** Incremental NFA evaluation

**(incNFA$^{\#}$)** Incremental NFA$^{\#}$ evaluation

For the other evaluation algorithms of Section 6.1, the cost of incremental evaluation will be the same as for evaluation from scratch so we don't include them again here. Our implementations of (incNFA) and (incNFA$^{\#}$) closely follows Section 5. Auxiliary data is stored as AA trees [1].

We use the following methodology to produce our measurements (and to compensate for the side-effects of garbage collection). Given a regular expression and a word, we first construct the auxiliary data structures to prepare for incremental evaluation. We then perform a large (and equal) number of insertions and deletions in the word; equally distributed at the beginning, middle, and end of the word. We then discard the 10% largest and 10% smallest measurements and take the average of the remaining ones.

We perform experiments of two kinds for increasing values of $n$: one lets the expression and the word grow simultaneously with $n$ and the other keeps the expression constant and only lets the word grow. As such we obtain a detailed insight in how the scalability depends on the word and on the expression.

**Expressions with Large Counter Values.** The leftmost graph in Fig. 5 contains our measurements for (incNFA) and (incNFA$^{\#}$) for expressions with large counter values as already described in Section 6.1: For a given value of $n$, we consider the expression $r = a^{(n-n/4),(n+n/4)}$ and we start with the word $w = a^n$. We then perform a large number of insertions and deletions and measure the average time of such an operation (as described before). Throughout the experiment, (incNFA$^{\#}$) outperforms (incNFA) and scales much better. Even for $n = 1,000,000$, (incNFA$^{\#}$) is faster than (incNFA) for $n = 50$; so, here, (incNFA$^{\#}$) is dealing with a word about five orders of magnitude larger than (incNFA). Lastly, recall that we do not have time measurements for (incNFA) for $n > 1000$ since preprocessing timed out. We also measured memory consumption (not in a separate chart for reasons of space) and noticed that, even for $n = 100$K, (incNFA$^{\#}$) consumes less memory than (incNFA) with $n = 1$K.

**Synthetic worst-case.** The second graph in Fig 5 depicts the incremental version of the corresponding experiment in Section 6.1. This test is designed to be hard on (incNFA$^{\#}$) even though expressions have large counter values. In this experiment the transformation tuples cannot summarize large sets of configurations of NFA$^{\#}$s as well as in the previous experiment, due to the non-determinism in the expressions. In terms of time, (incNFA$^{\#}$) consistently outperforms (incNFA) about one order of magnitude. In terms of space, (incNFA$^{\#}$) still scales much better than (incNFA); about two orders of magnitude.

**Real-world, MPEG7 Expression.** In this experiment and the next we consider a (fixed) expression that we found in our practical study and only let the length of the word grow as $n$ increases. This gives a clearer picture of the scalability of both evaluation methods when only the size of the data becomes very large. Here, we consider the expression $(ab^{2,12})^{0,65535}$ that we found in XML Schemas related to the MPEG-7 standard[14] and refer to as *MPEG7 expression*. From all the expressions we found in practice, this one seemed to be among the more challenging ones for state-of-the-art evaluation algorithms because it contains both nested counters and large counter values.

The MPEG7 expression is extremely challenging for (incNFA) because it translates into an NFA with about 850K states. As we can see from the third graph in Fig 5, incremental NFA evaluation for very short words (10 symbols) is already extremely slow. This is strongly contrasted by incremental NFA$^{\#}$ evaluation, which still evaluates incrementally in 0.08 milliseconds for words of length one million. We also say a huge difference in memory consumption: 5 orders of magnitude already for words of length 50.

**Real-world, CHAIN Expressions.** This final experiment is designed to capture the behavior of CHAIN-like expressions which we abundantly found in our practical study (Section 3). The experiment is extremely simple: we consider a fixed regular expression $a^{0,1000}$ (which we also found in our practical study) and incrementally evaluate with respect to words of increasing length $n$. We experimented with various and more complex forms of CHAIN expressions from our corpus, containing more disjunctions, more concatenations of disjunctions, larger alphabets, but we observed the

[14] http://standards.iso.org/ittf/PubliclyAvailableStandards/MPEG-7_schema_files/mpeg7-v2.xsd

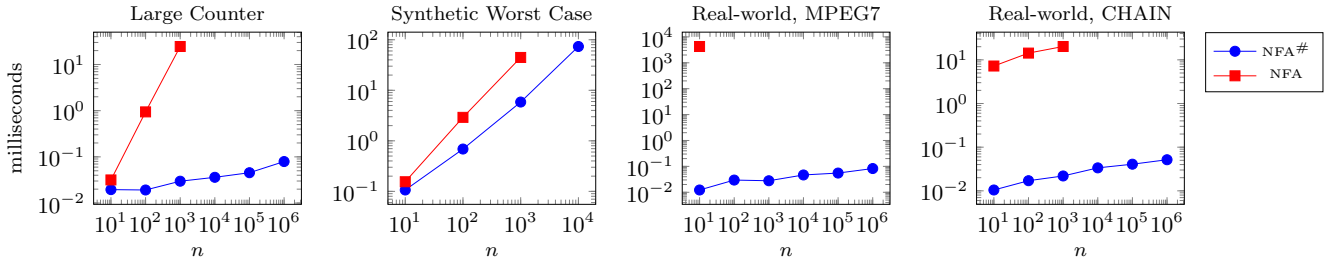



Figure 5: Incremental evaluation (Log-log scale).

same behavior in all cases. All experiments for CHAIN-like expressions boil down to this case. Again, as we can see the rightmost graph in Fig. 5, the evaluation algorithms based on NFA#s scale much better than those based on NFAs.

## 7. CONCLUSION

This paper shows that large counter values in regular expressions do not necessarily imply slow processing. Indeed, while regular expressions with counting often imply high complexity for static analysis problems due to their succinctness [12], it seems that the same succinctness can be exploited for designing highly efficient evaluation algorithms.

In the future, we want to extend the algorithm so that it works for full-fledged XML Schemas, make a more extensive study of regular expressions in Schemas and perhaps even in more general contexts, and look more closely at how to adapt the algorithms towards path expressions on graphs.

## 8. REFERENCES


[1] A. Andersson. Balanced search trees made simple. In *WADS*, pages 60–71, 1993.

[2] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM TODS*, 29(4):710–751, 2004.

[3] D. Barbosa, A.O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient incremental validation of XML documents. In *ICDE*, pages 671–682, 2004.

[4] M. Becchi and P. Crowley. Efficient regular expression evaluation: theory to practice. In *ANCS*, pages 50–59, 2008.

[5] M. Becchi and P. Crowley. A-DFA: A time- and space-efficient DFA compression algorithm for fast regular expression evaluation. *ACM TACO*, 10(1), 2013.

[6] G.J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML schema: a practical study. In *WebDB*, pages 79–84, 2004.

[7] H. Björklund, W. Gelade, and W. Martens. Incremental XPath evaluation. *ACM TODS*, 35(4):29, 2010.

[8] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Trans. Comput*, C-20(2):149–153, 1971.

[9] S. Dal-Zilio and D. Lugiez. XML schema, tree logic and sheaves automata. In *RTA*, pages 246–263, 2003.

[10] S. Gao, C.M. Sperberg-McQueen, and H. S. Thompson. W3C XML schema definition language (XSD) 1.1 part 1: Structures. Technical report, World Wide Web Consortium, April 2012.

[11] W. Gelade, M. Gyssens, and W. Martens. Regular expressions with counting: Weak versus strong determinism. *SIAM J. Comput.*, 41(1):160–190, 2012.

[12] W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. In *ICDT*, pages 269–283, 2007.

[13] G. Ghelli, D. Colazzo, and C. Sartiani. Linear Time Membership in a Class of Regular Expressions with Interleaving and Counting. In *CIKM*, 2008.

[14] V.M. Glushkov. The abstract theory of automata. *Russian Math. Surveys*, 16:1–53, 1961.

[15] Google custom search. www.google.com/cse.

[16] S. Harris and A. Seaborne. SPARQL 1.1 query language. Technical report, World Wide Web Consortium, January 2012. http://www.w3.org/TR/2012/WD-sparql11-query-20120105.

[17] D. Hovland. Regular expressions with numerical constraints and automata with counters. In *ICTAC*, pages 231–245, 2009.

[18] P. Kilpeläinen and R. Tuhkanen. Regular expressions with numerical occurrence indicators — preliminary results. In *SPLST*, pages 163–173, 2003.

[19] P. Kilpeläinen and R. Tuhkanen. Towards efficient implementation of XML schema content models. In *ACM DOCENG*, pages 239–241, 2004.

[20] K. Losemann and W. Martens. The complexity of regular expressions and property paths in SPARQL. *ACM TODS*, 2013.

[21] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *Siam J. Comp.*, 39(4):1486–1530, 2009.

[22] W. Martens, F. Neven, and S. Vansummeren. SCULPT: A schema language for tabular data on the Web. In *WWW Conference*, 2015. To appear.

[23] K. Nakayama, K. Yamaguchi, and S. Kawai. $\mathcal{I}$-regular Expression: Regular Expressions with continuous interval constraints. In *CIKM*, 1997.

[24] Neo4J. Cypher patterns. http://docs.neo4j.org/chunked/stable/introduction-pattern.html, 2014. Cypher Query Language, Section 8.8.

[25] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *JCSS*, 55(2):199–209, 1997.

[26] Regexlib. www.regexlib.com.

[27] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In *SIGCOMM*, pages 207–218, 2008.

[28] Snort. www.snort.org.

[29] C.M. Sperberg-McQueen. Notes on finite state automata with counters. http://www.w3.org/XML/2004/05/msm-cfa.html, 2004.

# APPENDIX

This appendix is not part of the submission and can be examined at the discretion of the reviewer. It provides further details for which there was no space in the body of the paper.

# A. PROOFS OF SECTION 5

We next argue that compositions of transformation tuples corresponds to our intended meaning and that they can be computed efficiently. The following lemma clarifies the semantics of tuple joins.

LEMMA A.1. *Let $t_1$ be a transformation tuple compatible with $t_2$ such that $t_1$ is consistent with the word $u$ and $t_2$ is consistent with $v$. Then $t_1 \bowtie_c t_2$ is consistent with $uv$.*

PROOF. Let $(p, \alpha)$ be a configuration such that $\alpha \models \phi$. We need to show that there is a configuration $(q, \alpha'')$ such that $(p, \alpha) \Rightarrow_{uv} (q, \alpha'')$, with $\alpha'' = \pi(\alpha)$.

From $\alpha \models \phi$ and Definition 5.5, we get $\alpha \models \phi_1$. Since $t_1$ is consistent with $u$, this means that there is a configuration $(r, \alpha')$, with $\alpha' = \pi_1(\alpha)$ such that $(p, \alpha) \Rightarrow_u (r, \alpha')$. Again, from $\alpha \models \phi$ and Definition 5.5, we get $\alpha' = \pi_1(\alpha) \models \phi_2$. Since $t_2$ is consistent with $v$, this means that there is a configuration $(q, \alpha'')$ with $\alpha'' = \pi_2(\alpha') = \pi_2(\pi_1(\alpha))$ such that $(r, \alpha') \Rightarrow_v (q, \alpha'')$. Since, by Definition 5.5, we have $\pi(\alpha) = \pi_2(\pi_1(\alpha))$, this gives us the desired result. □

The following lemma shows that the join of two transformation tuples can be computed very efficiently.

LEMMA A.2. *Given two transformation tuples $t_1$ and $t_2$, we can compute $t_1 \bowtie_c t_2$ in time linear in the number of counters of $t_1$ and $t_2$.*

PROOF. For the purposes of this proof, we use the following notation. If $t = (p, \phi, \pi, q)$ is a transformation tuple, $c \in C$ and $\phi(c) = (k, \ell)$, then we write $low(\phi(c))$ for $k$ and $up(\phi(c))$ for $\ell$. Also, if $\pi(c) = (action, k)$, then we write $\text{val}(\pi(c))$ for $k$ and $\text{type}(\pi(c))$ for *action*.

Let $t_1 = (p_1, \phi_1, \pi_1, q_1)$ and $t_2 = (p_2, \phi_2, \pi_2, q_2)$. We first need to check that $t_1$ and $t_2$ are compatible. Otherwise the join is undefined. Recall that $t_1$ and $t_2$ are compatible if $q_1 = p_2$ and there is a counter assignment $\alpha$ with $\alpha \models \phi_1$ and $\pi_1(\alpha) \models \phi_2$. The first condition is easy to check in constant time. For the second condition, we need to do one check per counter $c \in C$. If $\pi_1$ contains $\mathsf{assign}(c, k)$, for some $k$, then we only need to check that $low(\phi_2(c)) \leq k \leq up(\phi_2(c))$. If, on the other hand, $\pi_1$ contains $\mathsf{inc}(c, k)$ for some $k$, then we need to check that there exists a value $\alpha(c)$ such that $low(\phi_1(c)) \leq \alpha(c) \leq up(\phi_1(c))$ and $low(\phi_2(c)) \leq \alpha(c) + k \leq low(\phi_2(c))$. This holds if and only if $low(\phi_1(c)) + k \leq up(\phi_2(c))$ and $up(\phi_1(c)) + k \geq low(\phi_2(c))$. These checks can be performed in constant time.

For the remainder of this proof, let $t_1 = (p, \phi_1, \pi_1, r)$ and $t_2 = (r, \phi_2, \pi_2, q)$ be compaitble transformation tuples. Then $t_1 \circ_c t_2 = (p, \phi, \pi, r)$ is computed by, for every $c \in C$, determining the values of $low(\phi(c))$, $up(\phi(c))$, $\text{val}(\pi(c))$, and $\text{type}(\pi(c))$.

If $\text{type}(\pi_1(c)) = \mathsf{inc}$, then $\phi(c)$ is determined by Equations 1 and 2.

$$low(\phi(c)) = \\ = \max(low(\phi_1(c)), low(\phi_2(c)) - \text{val}(\pi_1(c))) \quad (1)$$

$$up(\phi(c)) = \\ = \min(up(\phi_1(c)), up(\phi_2(c)) - \text{val}(\pi_1(c))) \quad (2)$$

If, on the other hand, $\text{type}(\pi_1(c)) = \mathsf{assign}$, then $\phi(c) = \phi_1(c)$.

If $\text{type}(\pi_2(c)) = \mathsf{inc}$, then $\text{val}(\pi(c)) = \text{val}(\pi_1(c)) + \text{val}(\pi_2(c))$. If, on the other hand, $\text{type}(\pi_2(c)) = \mathsf{assign}$, then $\text{val}(\pi(c)) = \text{val}(\pi_2(c))$.

Finally, if both $\text{type}(\pi_1(c)) = \mathsf{inc}$ and $\text{type}(\pi_2(c)) = \mathsf{inc}$, then $\text{type}(\pi(c)) = \mathsf{inc}$. Otherwise, $\text{type}(\pi(c)) = \mathsf{assign}$.

Assuming unit cost arithmetic, these values can obviously be computed in constant time.

It remains to show that $t_1 \bowtie_c t_2$, thus computed, satisfies the two conditions from Definition 5.5. We first argue that the second condition holds, i.e., that $\pi(\alpha) = \pi_2(\pi_1(\alpha))$, for every assingment $\alpha$. We consider a counter $c \in C$. If $\text{type}(\pi_2(c)) = \mathsf{assign}$, then the effect of $\pi_2(\pi_1(\alpha))$ on $c$ will be to set $c$ to $\text{val}(\pi_2(c))$. In this case, $\text{type}(\pi(c)) = \mathsf{assign}$ and $\text{val}(\pi(c)) = \text{val}(\pi_2(c))$, so the effect of $\pi$ on $c$ will be the same. If $\text{type}(\pi_2(c)) = \mathsf{inc}$, then $\text{val}(\pi(c)) = \text{val}(\pi_1(c)) + \text{val}(\pi_2(c))$. If $\text{type}(\pi_1(c)) = \mathsf{assign}$, then the effect of $\pi_2(\pi_1(\alpha))$ on $c$ will be to set $c$ to $\text{val}(\pi_1(c))$ and then incrementing it by $\text{val}(\pi_2(c))$, effectively in total setting $c$ to $\text{val}(\pi(c))$, which is the effect $\pi$ will have on $c$. If $\text{type}(\pi_1(c)) = \mathsf{inc}$, then $\pi_2(\pi_1(\alpha))$ will increase $c$ by $\text{val}(\pi(c))$, again the same effect that $\pi$ will have on $c$.

We now turn to the first condition of Definition 5.5. Consider a counter assignmen $\alpha$ and assume that $\alpha \models \phi$. The definition of $\phi$ immediately gives us $\alpha \models \phi_1$, since for every $c \in C$, either $\phi(c) = \phi_1(c)$ or $\phi(c)$ is defined by Equations 1 and 2. In the latter case, the lower bound $\phi$ gives for $c$ is at least as high as the lower bound $\phi_1$ gives, and the upper bound $\phi$ gives for $c$ is at least as low as the upper bound $\phi_1$ gives.

If $\text{type}(\pi_1(c)) = \mathsf{inc}$, then $\pi_1(\alpha)(c) = \alpha(c) + \text{val}(\pi_1(c))$. The lower bound $low(\phi(c))$ that $\phi$ gives for $c$ is, by Equation 1, at least $low(\pi_2(c)) - \text{val}(\pi_1(c))$. By assumption, we have $low(\phi(c)) \leq \alpha(c)$. Thus we get

$$\pi_1(\alpha)(c) = \alpha(c) + \text{val}(\pi_1(c)) \geq \\ \geq low(\phi(c)) + \text{val}(\pi_1(c)) \geq low(\pi_2(c)).$$

A symmetrical argument shows that $\pi_1(\alpha)(c) \leq up(\pi_2(c))$ and we can conclude that $\pi_1(\alpha) \models \phi_2$.

If, on the other hand, $\text{type}(\pi_1(c)) = \mathsf{assign}$, then $\pi_1(\alpha)(c) = \text{val}(\pi_1(c))$. In this case, $low(\phi_2(c)) \leq \pi_1(\alpha)(c) \leq up(\phi_2(c))$ follows from the fact that $t_1$ is compatible with $t_2$. Indeed, if $\text{val}(\pi_1(c))$ was not an allowed value for $c$ according to $\phi_2$, there would be no assignment witnessing compatibility, as required by the definition.

For the other direction, assume that $\alpha \not\models \phi$. In particular, there must be a $c \in C$ such that $\alpha(c)$ violates $\phi$. Assume, without loss of generality, that $\alpha(c) < low(\phi(c))$. If $\text{type}(\pi_1(c)) = \mathsf{assign}$, then we immediately have $\phi(c) = \phi_1(c)$ and thus $\alpha(c) < low(\phi_1(c))$. If, on the other hand, $\text{type}(\pi_1(c)) = \mathsf{inc}$, then, according to Equation 1, we either have $low(\phi(c)) = low(\phi_1(c))$ or $low(\phi(c)) = low(\phi_2(c)) - \text{val}(\pi_1(c))$. In the first case, we again immediately have $\alpha(c) < low(\phi_1(c))$. In the second case, we use the fact that $\pi_1(\alpha)(c) = \alpha(c) + \text{val}(\pi_1(c))$ to get the inequality

$$\pi_1(\alpha)(c) = \alpha(c) + \text{val}(\pi_1(c)) < \\ < low(\phi(c)) + \text{val}(\pi_1(c)) = low(\phi_2(c)).$$

Thus we conclude that if $\alpha \not\models \phi$, then either $\alpha \not\models \phi_1$ or $\pi_1(\alpha) \not\models \phi_2$. This concludes the proof. □

Notice that the number of counters is bounded from above by the nesting depth of counters in expressions which is, in practice, very small, say, at most three (Section 3).

Our next step is to enable the use of transformation tuples in tables, just as pairs of states were used in tables in Section 5. We call such tables *transformation table*s.

DEFINITION A.3. A *transformation table* is a set of transformation tuples. A transformation table is *consistent* with respect to a string $w$ if every transformation tuple in the table is consistent with $w$. It is *complete* with respect to a string $w$ if, for every pair $\gamma = (p, \alpha)$ and $\gamma' = (p', \alpha')$ such that $\gamma \Rightarrow_w \gamma'$, there is a transformation tuple $t = (p, \phi, \pi, p')$ in the table with $\alpha \models \phi$ and $\alpha' = \pi(\alpha)$.

DEFINITION A.4. Let $T_1$ and $T_2$ be transformation tables. Then $\textsc{Join}(T_1, T_2)$ is the set

$$\{t_1 \bowtie_c t_2 \mid t_1 \in T_1, t_2 \in T_2, \text{ and } t_1 \text{ is compatible with } t_2\}.$$

Since we can join two single tuples in time linear in the number of counters according to Lemma A.2, we have that $\textsc{Join}(T_1, T_2)$ can be computed in time $\mathcal{O}(|T_1| \times |T_2| \times |C|)$, where $|T_i|$ is the number of tuples in table $T_i$ for each $i = 1, 2$. (This bound can be obtained by a simple nested loop join. In our implementation, however, we use a faster join algorithm.) The following lemma states that our procedure for joining transformation tables is correct.

LEMMA A.5. *If transformation tables $T_u$ and $T_v$ are complete and consistent with respect to the strings $u$ and $v$, respectively, then $T_{uv} = \textsc{Join}(T_u, T_v)$ is complete and consistent with respect to $uv$.*

PROOF. Consistency follows immediately from the definition of the table join and Lemma A.1.

To show compelteness, let $\gamma = (p, \alpha)$ and $\gamma' = (q, \alpha')$ be such that $\gamma \Rightarrow_{uv} \gamma'$. We need to show that there is a tuple $t = (p, \phi, \pi, q)$ in $\textsc{Join}(T_u, T_v)$ such that $\alpha \models \phi$ and $\alpha' = \pi(\alpha)$.

Let $\gamma'' = (r, \alpha'')$ be the configuration $A$ takes after reading $u$ in the unique run on $uv$ startin from $\gamma$. Since $T_1$ is complete with respect to $u$, there must be a tuple $t_1 = (p, \phi_1, \pi_1, r)$ in $T_1$ such that $\alpha \models \phi_1$ and $\alpha'' = \pi_1(\alpha)$. Also, since $T_2$ is complete with respect to $v$, there must be a $t_2 = (r, \phi_2, \pi_2, q)$ in $T_2$ such that $\alpha'' \models \phi_2$ and $\alpha' = \pi_2(\alpha'')$. This means that $t_1$ and $t_2$ are compatible, and thus $t_1 \circ_c t_2$ is a tuple in $T_{uv}$. Let this be the tuple $(p, \phi, \pi, q)$. From Definition 5.5 we get that $\pi(\alpha) = \pi_2(\pi_1(\alpha)) = \alpha'$. Also, from the same definition and since $\alpha \models \phi_1$ and $\alpha'' = \pi_1(\alpha) \models \phi_2$, we have $\alpha \models \phi$. □

```
    function JoinEntries(Entry t1, Entry t2)
 2:   if t1.endState ≠ t2.startState then
        return NULL
 4:   end if
      t ← new Entry
 6:   for Counter c ∈ C do
        t.isIncrement(c) ← t1.isIncrement(c) ∧
    t2.isIncrement(c)
 8:     if t1.isIncrement(c) then
          t.low(c) ← max(t1.low(c), t2.low(c) − t1.incVal(c))
10:       t.upp(c) ← min(t1.upp(c), t2.upp(c) − t1.incVal(c))
          if t.low(c) > t.upp(c) then
12:         return NULL
          end if
14:       if t2.isIncrement(c) then
            t.incVal(c) ← t1.incVal(c) + t2.incVal(c)
16:       else
            t.writeVal(c) ← t2.writeVal(c)
18:       end if
        else
20:       if t2.low(c) > t1.writeVal(c)
                or t2.upp(c) < t1.writeVal(c) then
22:         return NULL
          end if
24:       t.low(c) ← t1.low(c)
          t.upp(c) ← t1upp(c)
26:       if t2.isIncrement(c) then
            t.writeVal(c) ← t1.writeVal(c) + t2.incVal(c)
28:       else
            t.writeVal(c) ← t2.writeVal(c)
30:       end if
        end if
32:   end for
      t.startState ← t1.startState
34:   t.endState ← t2.endState
      return t
36: end function
```

Figure 6: Computing $t_1 \circ_c t_2$ for given $t_1$ and $t_2$. The algorithm uses a more object-oriented notation than, e.g., the proof of Lemma A.2. For example, if $t_1 = (p, \phi_1, \pi_1, r)$, then in the algoirthm, we use $t_1$.endState for $r$, $t_1$.isIncrement($c$) to check if type($\pi(c)$) is inc, $t_1$.low for $low(\phi_1(c))$, incVal for a value with which a counter should be increased, writeVal for a value which a counter should be assigned, etc.

```
    function Join(Table T1, Table T2)
 2:   Table T ← ∅
      for t1 ∈ T1 do
 4:     for t2 ∈ T2 do
          t ← JoinEntries(t1, t2)
 6:       if t ≠ NULL then
            T.addEntry(t)
 8:       end if
        end for
10:   end for
      return T
12: end function
```

Figure 7: Joining two transformation tables $T_1$ and $T_2$.