# Simplifying Regular Expressions Further

Stefan Kahrs[a,*], Colin Runciman[b]

[a] *School of Computing, University of Kent, Canterbury CT2 7NF, UK*
[b] *Department of Computer Science, University of York, York YO10 5GH, UK*

**Abstract**

We describe a cumulative series of transformations to simplify regular expressions, and investigate their effectiveness and cost. Transformations depending on increasingly powerful comparisons of expressions give results clearly superior to commonly used algebraic simplifications. Early in the series, efficient transformations enabled by language-invariant attributes are surprisingly effective. Later in the series, transformations depending on comparisons of expressed languages are made feasible by bounding the size of subexpressions to which they are applied. We set out the principles of our transformations, address some key implementation issues, and evaluate the results of systematic test measurements.

*Keywords:* Regular Expression, Algebraic Simplification, Semantic Simplification,
*2000 MSC:* 03D40

## 1. Introduction

Regular expressions (hereafter "expressions") were introduced by Kleene around seventy years ago [1] as concise specifications of regular languages. Where many expressions all specify the same language, a smallest expression is usually easiest to understand and costs the least to store and process.

Finite-state acceptors (FSAs) also specify regular languages, and there is a long-established algorithm [2] for simplification of a deterministic FSA to a unique minimal form. More recently, there has also been significant progress on the problem of reducing non-deterministic FSAs [3]. However, although various algebraic laws about expressions [4] are routinely used as simplification rules, there is no efficient equivalent to FSA minimisation for expressions.

Using only algebraic laws between terms without repeated variables avoids comparison costs and allows simplification as a whole to be linear-time. One good representative for this approach is the simplifying transformation of Gruber & Gulan [5]. Various textbooks, on-line teaching notes and web-based simplifiers approach the problem in a similar way, shying away from computationally harder methods, and avoiding a quantitative evaluation.

At the extreme end the problem is, of course, recursive: we can enumerate expressions in order of size, compare them semantically with the input expression, and the first equivalent expression we find is our answer. This is the "extreme end" because the semantic comparison is PSPACE-hard [6], and the number of expressions to be considered grows exponentially in relation to the size of the input expression.

In this paper we explore a range of transformation methods. We show how to achieve significantly better results than the usual algebraic simplifications alone. A staged approach makes it possible to control the additional cost of more powerful methods.

## 2. Definitions

Syntactically, given some finite alphabet $\Sigma$, the forms of expression we adopt are 0, 1, $s$, $x + y$, $x \cdot y$, $x^*$, $x?$ and $(x)$ where $s \in \Sigma$ and $x$, $y$ are any finite expressions. Unless overridden by brackets, the binding priorities are first the unary operators $*$ (Kleene star) and ? (option), then $\cdot$ (sequence), then $+$ (choice). As $\cdot$ and $+$ are associative, no brackets are required in $\cdot$ or $+$ chains — as we explain further in §4, sequence and choice are both N-ary in our implementation.

Semantically, any expression $x$ specifies a language $L(x)$, a set of words over alphabet $\Sigma$. If $x$, $y$ are expressions the usual meaning of $x = y$ is $L(x) = L(y)$. Concerning atomic expressions: $L(0) = \emptyset$ the empty set, $L(1) = \{\epsilon\}$ where $\epsilon$ is the empty word, and $L(s) = \{s\}$. Concerning compound expressions:

$$L(x + y) = L(x) \cup L(y)$$
$$L(x \cdot y) = \{vw \mid v \in L(x), w \in L(y)\}$$

and the semantics of ? and $*$ are given by $x? = 1 + x$ and the least solution to $x^* = 1 + x \cdot x^*$. Although the ? operator adds no expressive power, it is notationally convenient, and it is useful in the formulation and implementation of our simplifying transformations.

Some laws are true only if an expression has the *empty word property*. We define $ewp(x) \iff \epsilon \in L(x)$.

*Corresponding author: S.M.Kahrs@kent.ac.uk

We need a few other language-related attributes of an expression $x$. These include the sub-alphabets of symbols occurring leftmost, rightmost, or anywhere, in any word of $L(x)$; also those occurring as single-letter words. We define

$$\begin{aligned}\overleftarrow{\alpha}(x) &= \{s \mid sw \in L(x)\}\\ \overrightarrow{\alpha}(x) &= \{s \mid ws \in L(x)\}\\ \alpha(x) &= \{s \mid vsw \in L(x)\}\\ \alpha_1(x) &= \{s \mid s \in L(x)\}\end{aligned}$$

We also define related functions whose results are expressions. If $f(x) = \{s_1, ..., s_n\}$ then $\widehat{f}(x) = s_1 + \cdots + s_n$. So, for example, the *total language* of all words over $\alpha(x)$ is specified by the expression $\widehat{\alpha}(x)^*$.

The *size* of expressions is of special interest in this paper. We define size$(x)$, a non-negative integer:

$$\begin{aligned}\text{size}(0) = \text{size}(1) &= 0\\ \text{size}(s) &= 1\\ \text{size}(x + y) = \text{size}(x \cdot y) &= \text{size}(x) + 1 + \text{size}(y)\\ \text{size}(x^*) = \text{size}(x?) &= \text{size}(x) + 1\end{aligned}$$

The zero sizes of 0 and 1 are justified by their easy elimination as parts of larger expressions (see §3.1). Appropriately, size$(x?)$ = size$(1 + x)$. We say an expression $x$ is *minimal* if there is no expression $y$ for which $y = x$ and size$(y) <$ size$(x)$.

## 3. Simplifying Transformations

We have investigated a hierarchy of layered transformations. We discuss them from the simplest to the most sophisticated, and explain what laws or methods are characteristic for each transformational layer. We also briefly explain the expected time-complexity of a layer when applied to expressions of size $s$.

We generally apply laws as size-reducing transformations. However, laws that leave size unchanged are also used to obtain particular normal forms. Such size-preserving transformations are generally not applied backwards, except in composition with a size-reducing law, for example $x^* \cdot (x? \cdot z) = (x^* \cdot x?) \cdot z = x^* \cdot z$.

### *3.1. Standardisation*

Standardisation realises a sub-theory of the algebraic laws on expressions [4]. It computes a *quotient* for this sub-theory, giving a unique *normal form* for each equivalence class. Its time-complexity is $O(s \log(s))$, mostly incurred by the sorting of alternatives. Our standardisation sub-theory consists of the choice laws

$$\begin{aligned}x + y &= y + x\\ x + (y + z) &= (x + y) + z\\ x + x = x + 0 &= x\\ x + 1 &= x?\\ x + y? &= (x + y)?\end{aligned}$$

the sequence laws

$$\begin{aligned}x \cdot (y \cdot z) &= (x \cdot y) \cdot z\\ 1 \cdot x = x \cdot 1 &= x\\ 0 \cdot x = x \cdot 0 &= 0\end{aligned}$$

the star laws

$$\begin{aligned}0^* = 1^* &= 1\\ (x^*)^* = x?^* &= x^*\end{aligned}$$

and the option laws

$$\begin{aligned}0? = 1? &= 1\\ x? &= x \ \textbf{if} \ \text{ewp}(x)\end{aligned}$$

The first, third and last laws need further explanation. About the first and third, *commutativity* and *idempotence* of $+$: in a standard-form choice $x_1 + \ldots + x_n$ none of the $x_i$ is itself a choice, and $\forall i < n.\, x_i \prec x_{i+1}$ where $\prec$ is some linear order on expressions — see §4 for implementation details. About the last law, which holds conditionally: we exploit it not only from left to right, to make an expression smaller, but also from right to left: whenever we write a rule with an explicitly optional sub-expression $x?$, the rule also applies for any sub-expression $x$ with ewp$(x)$.

As an example of standardisation:

$$\begin{aligned}& b? + a \cdot 1 + a?\\ \rightarrow\ & \{\text{sequence law: } x \cdot 1 = x\}\\ & b? + a + a?\\ \rightarrow\ & \{\text{choice laws: } \cdots + x? + \cdots = (\cdots + x + \cdots)?\}\\ & (b + a + a)??\\ \rightarrow\ & \{\text{option law: } x? = x \text{ when ewp}(x)\}\\ & (b + a + a)?\\ \rightarrow\ & \{\text{choice is associative, commutative \& idempotent}\}\\ & (a + b)?\end{aligned}$$

Although we only appeal explicitly to commutativity and associativity of choice in the final step, in practice they are implicit throughout as we use an N-ary representation and a standard ordering. In particular, they are implicit in the option-lifting second step.

### *3.2. Fusion*

Fusion uses structural comparison to eliminate redundant parts of an expression, according to these rules:

$$\begin{aligned}x^* \cdot x^* = x^* \cdot x? = x? \cdot x^* &= x^*\\ (x? \cdot y?)^* = (x + y?)^* = (x + y^*)^* &= (x + y)^*\end{aligned}$$

These laws imply that any expression of the form $x^*$ can be expressed as $y^*$ where size$(y) \leq$ size$(x) \wedge \neg$ewp$(y)$ [5].

For example:

$$\begin{array}{ll} & (a^* \cdot b?)^* \\ = & \{\text{option introduction: } x = x? \text{ when ewp}(x)\} \\ & (a^*? \cdot b?)^* \\ = & \{\text{fusion law: } (x? \cdot y?)^* = (x+y)^*\} \\ & (a^* + b)^* \\ = & \{\text{fusion law: } (x + y^*)^* = (x+y)^*\} \\ & (a+b)^* \end{array}$$

For this example we have used equational reasoning, with an implicit appeal to commutativity in the final step. Using *context-aware* transformations we can implement steps such as fusion under $*$ more directly — see §4.1.

Like standardisation, fusion has $O(s \log(s))$ time complexity. It includes a factorisation method that can be seen as a modified merge-sort applied to choices. This is realised by using two different linear orders under which expressions with common prefix, or suffix, are adjacent to one another. In each merge operation, comparison of expressions may lead to their "fusion" as both are removed and combined into a new expression using the distributive laws:

$$x \cdot y + x \cdot z = x \cdot (y + z)$$
$$x \cdot z + y \cdot z = (x + y) \cdot z$$

Because any new expression has the same prefix or suffix as the expressions it was built from, the process can continue, without re-ordering, to find any further factorisations. For example, fusion simplifies:

$$\begin{array}{ll} & a \cdot a^* + a^* \\ \rightarrow & \{\text{fusion, factorisation}\} \\ & (a+1) \cdot a^* \\ \rightarrow & \{\text{standardisation}\} \\ & a? \cdot a^* \\ \rightarrow & \{\text{fusion, sequence laws}\} \\ & a^* \end{array}$$

However, fusion does *not* implement a quotient. Fusion can have different normal forms for the same equivalence class. For example, the expression $a \cdot (b + c + d) + b \cdot (c + d)$ is a normal form for fusion of size 13. Yet it is equivalent under the distributive laws to $a \cdot b + (a + b) \cdot (c + d)$, another normal form of fusion that is only of size 11.

### *3.3. Lifting*

Lifting exploits several language-invariant attributes of expressions. In particular, it identifies sub-expressions $x$ for which $L(x)$ includes the total language over some sub-alphabet of $\alpha(x)$. Lifting uses this information to absorb choice alternatives or sequence items. It also uses language attributes to enhance factorisation. As we shall see, the time complexity of lifting is $O(s^2)$.

How do we identify when an expression specifies the total language over its alphabet? For expressions of the form $x^*$ this is solely determined by the following principle:

$$x^* = \widehat{\alpha}_1(x)^* \ \textbf{if} \ \alpha(x) = \alpha_1(x)$$

Even when $L(x^*)$ is not the total language over $\alpha(x)$, it may include the total language over a sub-alphabet. We have the following generalisation of the previous law:

$$(x + y)^* = (\widehat{\alpha}_1(x) + y)^* \ \textbf{if} \ \alpha(x) \subseteq \alpha_1(x + y).$$

For example, by the first law we can simplify

$$(a + a? \cdot b)^* \ \rightarrow \ (a + b)^*$$

and by the second law we can simplify

$$(b \cdot c + a? \cdot (b + c))^* \ \rightarrow \ (0 + a? \cdot (b + c))^* \ \rightarrow \ (a? \cdot (b + c))^*$$

Further, when total-language expressions occur in sequences, they may be able to absorb adjacent expressions. For example, we have the rules:

$$x^* \cdot y? = x^* \ \textbf{if} \ \alpha(y) \subseteq \alpha_1(x)$$
$$(x \cdot y?)^* = x^* \ \textbf{if} \ \alpha(y) \subseteq \alpha_1(x)$$

For choices, we also have a subsumption rule:

$$x^* + y = x^* \ \textbf{if} \ \alpha(y) \subseteq \alpha_1(x)$$

Inclusion of this rule makes lifting a worst-case quadratic-time transformation, because in an $n$-ary choice $x_1 + \cdots + x_n$ each pair $(x_i, x_j)$ could potentially be an instance of $(x^*, y)$.

Typically, lifting laws are special cases of more general laws that can be expressed in terms of the equality or inclusion of languages. For example, we could replace the condition in each of the last three laws by $L(y) \subseteq L(x^*)$. However, language inclusion and equality are PSPACE-complete problems [6], whereas the application conditions for lifting are checked in constant time in our implementation.

### *3.4. Pressing*

Pressing transformations use the full power of language comparison, both for equivalence and inclusion. Pressing applies similar laws to those used for lifting, but with more general conditions. Without size-constraints pressing would be PSPACE-complete; with size-constraints it is PTIME, which derives from the selection of size-bounded sub-expressions in choices, modulo standardisation.

There are various pressing rules for $+$ subsumption, either where $x + y = x$, or where $x + y = x + y'$ and $\text{size}(y') < \text{size}(y)$. To find instances of the second kind, pressing tries to construct an expression $y'$ such that $L(y) \setminus L(x) \subseteq L(y') \subseteq L(x + y)$. We write $y \searrow x$ for an expression $y'$ with this property. Related techniques have been

applied previously to non-deterministic FSAs [3], for *transition pruning* and *transition saturation*. Here, we increase or decrease the language of a sub-expression rather than a state.

A useful partial approximation of $\searrow$ can be defined by induction on sub-expressions. There is not room here to give in full our rules for $\searrow$, but by way of illustration special cases include:

$$x? \searrow y = x \ \textbf{if} \ \mathrm{ewp}(y)$$
$$x \cdot y \searrow z^* = x \cdot (y \searrow z^*) \ \textbf{if} \ L(x) \subseteq L(z^*)$$

For example, pressing finds the simplification:

$$a{\cdot}b?{+}a^* \ \rightarrow \ (a{\cdot}b? \searrow a^*){+}a^* = a{\cdot}(b? \searrow a^*){+}a^* = a{\cdot}b{+}a^*$$

Other pressing rules use language comparison to extend the reach of factorisation, enabling further simplifications by the distributive law. For example:

$$a{\cdot}(b{\cdot}a)^*{+}(a{\cdot}b)^*{\cdot}b = (a{\cdot}b)^*{\cdot}a{+}(a{\cdot}b)^*{\cdot}b \rightarrow (a{\cdot}b)^*{\cdot}(a{+}b)$$

Finally, pressing also uses language comparison to support *star introduction*. If we find a sub-expression $x$ for which $L(x) = L(x^*)$, introducing the star may seem an odd move. However, after fusion the result can never be larger than $x$, and it may admit further simplification by lifting or by other pressing rules. For example, pressing has the rule:

$$x? \cdot y? = (x + y)^* \ \textbf{if} \ L(x? \cdot y?) = L((x? \cdot y?)^*)$$

So pressing searches for (sub-)sequences $x = x_1 \cdot \ldots \cdot x_k$ for which $\mathrm{ewp}(x)$ and $L(x^*) = L(x)$. If found, it replaces $x$ by $(x_1 + \ldots + x_k)^*$ which fusion will simplify to an expression of smaller size than $x$. So, for instance:

$$\begin{array}{ll} & (a \cdot a)^* \cdot a? \\ \rightarrow & \{\text{pressing rule above}\} \\ & ((a \cdot a)^* + a)^* \\ \rightarrow & \{\text{fusion including factorisation}\} \\ & (a \cdot a?)^* \\ \rightarrow & \{\text{lifting}\} \\ & a^* \end{array}$$

### *3.5. Syntactic lookup*

After we have done our utmost using algebraic transformation rules, we may still hope to find further simplifications by resorting to *catalogues* of all minimal expressions within specified bounds. Our final simplification stages seek matches of size-bounded expressions in such catalogues. Like pressing, and for the same reasons, these stages have PTIME complexity.

The syntactic catalogue is a collection of finite maps, for different alphabet sizes, each mapping pressed expressions up to a certain size limit to their minimal form. The catalogue is built by enumerating the set of all pressed expressions up to that size, partitioning this set by language equivalence, and picking a smallest representative in each equivalence class as the target value to which others are mapped.

Catalogue mappings only include cases where the size of the expression is strictly reduced. We exclude size-preserving cases, even though they could help with factorisation, because catalogue lookup is by structural comparison modulo renaming (see §4), and the combination of renaming and size-preserving rules can introduce non-termination.

When trying to simplify an expression by syntactic lookup, the entire expression is looked up if it is within the size-bound for its alphabet. Otherwise its largest sub-expressions within bounds are looked up. Here we include as sub-expressions both sub-sequences of sequence expressions and sub-choices of choice expressions, as they are obtainable as subterms by the algebraic laws of standardisation.

For example, the syntactic catalogue includes this simplification:

$$a + b \cdot b^* \cdot a? \rightarrow b^* \cdot (a + b)$$

Pressing misses this simplification, because algebraically we first have to make the expression bigger before we can make it smaller: $a{+}b{\cdot}b^*{\cdot}a? = a{+}b{\cdot}b^*{\cdot}(a{+}1) = a{+}b{\cdot}b^*{\cdot}a{+}b{\cdot}b^* = (1{+}b{\cdot}b^*){\cdot}a{+}b{\cdot}b^* = b^*{\cdot}a{+}b^*{\cdot}b = b^*{\cdot}(a{+}b)$.

### *3.6. Semantic lookup*

The semantic catalogue represents a family of linear-ordered sets, for different alphabet sizes, of all regular languages specifiable by expressions up to a certain size limit.

Each language is represented by a minimal expression that specifies it. Lookup is by a refinement of language comparison to a linear order (see §4). As the catalogue is a collection of languages, to find a minimal equivalent of expression $x$, we search the catalogue by linear-order comparisons with $L(x)$. If the catalogue contains $n$ languages for alphabet size $|\alpha(x)|$, then we can find a minimal expression for $L(x)$, if it is represented in the catalogue, by $O(\log n)$ language comparisons.

Like syntactic lookup, semantic lookup involves renaming, for example renaming in $x$ with $|\alpha(x)| = 3$ to obtain an expression over the alphabet $\{a, b, c\}$. Here too, not only an expression itself but also its sub-expressions are candidates for look-up in the catalogue. In contrast to the syntactic catalogue, the size of an expression does not rule out its representation in the catalogue, but the size of its alphabet might do so.

For example, here is a simplification by semantic lookup which all previous phases, including syntactic lookup, fail to find:

$$(a? \cdot b \cdot (a^* \cdot b^* \cdot a)?)^* \ \rightarrow \ (a? \cdot b \cdot a^*)^*$$

The left-hand expression is too large for the syntactic catalogue, but the right-hand expression is small enough to be in the semantic catalogue.

### 3.7. Minimisation

Finally, there is the "extreme end" of the transformation spectrum, mentioned in the introduction. We can enumerate expressions in order of increasing size until eventually the first equivalent to a given expression is found. Of course the cost may be prohibitive. To reduce it somewhat, the input can first be simplified by some transformation $T_1$, and the enumerated search-space can be confined to normal forms of some transformation $T_2$. Even with such improvements, we have found minimisation is hardly usable beyond expressions over unary alphabets.

## 4. Implementation

We have implemented all the transformations of §3 in the functional language Haskell[7].

The algebraic datatype for expressions represents them as trees. N-ary sequence and choice nodes both carry as memoised values the size of the sequence or choice as an expression, and its language attributes *ewp*, $\overleftarrow{\alpha}$, $\overrightarrow{\alpha}$, $\alpha$ and $\alpha_1$.

Any order-based implementation of sets could be used to represent N-ary choices. We simply use ordered lists without duplicates. Any convenient cheaply-computed linear order over expressions would do. We use a straightforward structural ordering for the expression type. The cost of this ordering is $O(q)$, where $q$ is the size of the smaller expression; so sorting $p$ sub-expressions of size $q$ is $O(p\log(p)q)$, and as $pq$ is bounded by the size $s$ of the whole expression, sorting is $O(s\log(s))$ overall.

Syntactic comparison of expressions, used for example to order choices, is at worst linear. Semantic comparisons use methods based on Brzozowski derivatives [8]. The implementation of our linear order on languages is similar to Almeida's language equivalence test [9]; although it has exponential complexity in the worst case, this seldom arises in practice.

### 4.1. Transformations in Context

Transformations are expressed as *context-aware* rewrite rules that apply to sequences or choices. These rules are parametrised by the context of the expression to which they are applied. Possible contexts are the free context, the optional context, or the repetition context. For example, to transform an expression $x$ in the repetition context we seek a replacement $y$ such that $x^* = y^*$. Analogously, in the optional context, $y$ has to satisfy $x? = y?$.

Contexts arise not only for direct sub-expressions such as $x$ in $x^*$. For example, if we transform the sub-expression $x$ in $(x + z)^*$ then it can be transformed in the repetition context: if $x^* = y^*$ then $(x+z)^* = (x^*+z)^* = (y^*+z)^* = (y + z)^*$. Optional contexts also arise in other ways. For example, $x$ in $x + z^*$ can be transformed in the optional context.

Contexts are hierarchical: any transformation applicable in the free context is applicable in the optional context, any transformation applicable in the optional context is also applicable in the repetition context. Often a transformation in a stronger context is just a stronger variant of a transformation in a weaker one.

For example, when considering whether $x + y$ can be simplified to $x$ we need to check $L(y) \subseteq L(x)$; but when transforming $x+y$ in an optional or repetition context the check can be weakened to $L(y) \subseteq L(x?)$ or $L(y) \subseteq L(x^*)$, respectively. A traditional bottom-up algebraic approach would transform $(x + y)^*$ by first transforming $x + y$ and then transforming the result starred. This entails first checking the stronger condition then, if it fails, the weaker one. Instead, we make the transformations context-aware, only checking the weaker condition.

### 4.2. Transformation Tags

Suppose a transformational layer $T$ has been exhaustively applied to an expression in context $c$, with expression $x$ as result. Then we tag $x$ as a $(T, c)$-normal form. If other rules are subsequently applied to an expression containing $x$, and $x$ is preserved, then it retains all its normal-form tags. However, tags of higher transformation layers or stronger contexts subsume those of lower layers or weaker contexts.

Generally, we first transform the whole expression with a lower-layer transformation, tagging it throughout in the process, and then proceed to transformations with the next layer. For various reasons, we prefer this strategy to a sub-expressions-first strategy. For example, if lifting finds $x = \alpha(x)^*$ then any effort transforming sub-expressions of $x$ would most likely be wasted, as lifting is based on language-invariant attributes.

Tags can also be inherited when forming sub-sequences of sequences or sub-choices of choices. For example, suppose we factorise an expression of the form $x \cdot y + b \cdot y \to (x + b) \cdot y$ in which $x$ is a sequence. In the original expression, since $x$ is a sub-sequence it has no tags of its own; but after factorisation it can inherit the free-context tags of $x \cdot y$.

Special tags of the form $(\mathit{minimal}, c)$ are awarded when it is known that an expression is minimal in context $c$. Minimality subsumes all other transformational tags, enabling us to avoid redundant searches for applicable simplifications. Minimal tags may arise from a successful catalogue look-up. Also, any sequence of symbols, and any choice of distinct symbols, is directly tagged as minimal when we form a standardised expression.

### 4.3. Divide and Conquer

Some transformations do not scale well. So we seek ways to break larger expressions into parts and transform them independently. When transforming expressions of the forms $x^*$ or $x?$ we simply transform their body $x$ in the appropriate context. In particular, any starred expression always has a starred minimal equivalent.

A similar principle applies to sequence expressions starting or ending with a single symbol. An expression $s \cdot x$ is

minimal in a free context if and only if $x$ is. So to transform an expression of the form $s \cdot x$ in a free context, we may transform $x$ to $x'$ and then lift the transformation tag of $x'$ to the whole expression $s \cdot x'$. This can lead to stronger tags than those of the transformation we are currently applying: for example, the expression $a \cdot (a+b)^* \cdot b$ can be tagged as *minimal* in a free context.

Whenever $\alpha(x) \cap \alpha(y) = \emptyset$, $x$ and $y$ can be transformed independently within $x \cdot y$ or $x + y$. There is a proviso for choices: if $\mathrm{ewp}(x)$ (or $\mathrm{ewp}(y)$) then the transformation context of $y$ (or $x$) is at least optional. Unless the transformation context of $x + y$ is a repetition, we can do better: for independent transformation of $x$ and $y$ it is enough that both $\overleftarrow{\alpha}$ and $\overrightarrow{\alpha}$ give disjoint results for $x$ and $y$. When transforming n-ary choices $x_1 + \cdots + x_n$ we try to partition the $x_i$ into groups accordingly. For example, $a \cdot b^* + c \cdot b \cdot c + a? \cdot b$ can be partitioned into $a \cdot b^* + a? \cdot b$ and $c \cdot b \cdot c$.

### *4.4. Size bounding*

During development we found that later transformations of pressing and catalogue lookup were too expensive for unrestricted application to large expressions. So by default we limit the size of sub-expressions to which they are applied. Candidate sub-expressions include segments of oversized sequences and subsets of oversized choices.

Our default limits for the size of expression to which pressing transformations are applied are 15 for choices and 20 for sequences. Raising these limits slightly improves simplification but greatly increases run-times. For example, if we raise the limits to 20 and 25, the improvement is tiny for large inputs but typically run-time increases by around 25% — far more in the worst case.

As catalogues are stored in files, they are of limited size and limited alphabet size. Specifically, for alphabet sizes from 1 to 4, we set expression-size limits of 15, 12, 11 and 10 for the syntactic catalogue, and 15, 11, 9 and 8 for the semantic catalogue. This means, for example, that all languages over a singleton alphabet representable by an expression of size up to 15 are in the semantic catalogue, and all expressions of up to that size also have a minimal equivalent in the syntactic catalogue. In all there are 24,077 entries in the syntactic catalogue, and 28,478 languages in the semantic catalogue.

The time to generate the catalogues grows exponentially with these expression-size limits, because the pool of expressions from which they are built grows exponentially. Raising the expression-size limits for catalogues by just 1, for each alphabet size, approximately quadruples the overall size of each catalogue (to 98,620 entries for the syntactic catalogue and to 91,414 entries for the semantic catalogue). Here again there is only a very slight improvement in simplification.

### *4.5. Renaming*

We cannot directly lookup expressions "modulo renaming" in either the syntactic or the semantic catalogue, as neither our structural comparisons nor our linear order on languages is preserved by renaming. So we need to rename expressions first to match the alphabets used in the catalogues. One option would be to create a unique catalogued representative for each class of renaming-equivalent expressions. However, lookup would then be very costly, iterating over many possible renamings. In general, if $|\alpha(x)| = n$ then $x$ has $n!$ renamings. For example, $a \cdot b \cdot c? + d$ has 24 renamings.

Instead, we use intermediate solutions, storing small subsets of renaming equivalents in catalogues. For the semantic catalogue, any class of renaming-equivalent expressions can be partitioned into subsets of expressions with identical memoised alphabets $\overleftarrow{\alpha}$, $\overrightarrow{\alpha}$ and $\alpha_1$. This can be viewed as the base of a topology on expressions, and the renamings we construct target just one of those classes and have to be continuous w.r.t. that topology. Concretely this is done by bringing distinguishable elements of $\alpha(x)$ into a particular order.

Returning to the example, the semantic catalogue contains just two of the 24 renamings of $a \cdot b \cdot c? + d$. These two are needed as they occupy the same target class for the renaming. They are $a + b \cdot c \cdot d?$ and $a + b \cdot d \cdot c?$.

## 5. Evaluation

We determine the effectiveness of our transformations by systematic testing and measurement. When a size-reducing transformation $T$ is applied to an expression $x$ one simple and natural measure of its effectiveness is the ratio $\mathrm{size}(T(x))/\mathrm{size}(x)$. This measure is readily extended to a test-set $X$ of expressions by computing the geometric mean:

$$\operatorname*{GM}_{x \in X} \left( \frac{\mathrm{size}(T(x))}{\mathrm{size}(x)} \right)$$

### *5.1. Effectiveness for Random Samples of expressions*

We shall first present results for randomly generated samples of expressions. As a general observation, when expression size is fixed, increasing alphabet size increases the probability that a random expression is minimal, converging to 1 in the limit. This is so, simply because an expression with at most one star or option and all symbols distinct is necessarily minimal. Consequently, the effectiveness of transformations on random expressions inevitably diminishes with increased alphabet size.

For a fixed alphabet and target expression size there are only finitely many different expressions, but even for quite modest sizes there are too many expressions for exhaustive testing to be feasible. So we pick samples of 1000 expressions at random, with uniform distribution. We have found this sample size is large enough for different runs to give very similar results.

Each sample is drawn from a population of expressions with specified size and alphabet size. We exclude expressions with 0 or 1 as sub-expressions to avoid a predominance of easy targets.

The method for generating random expressions is taken from [10]. Before generating expressions of a given size and alphabet size, we determine by combinatorial analysis the relative proportions of possible decompositions of such expressions. The generation process uses these results to attach probabilities to different operator symbols.

As we want our tests to include expression sizes in the thousands, computations with unbounded integers are too slow, and standard floating point numbers run out of exponent space; so we represent population counts as floating-point numbers with extra-large exponents.

The bar charts in Figures 1–3 summarise mean-size-ratio results for expressions over alphabets of sizes 2, 4 and 8 respectively. Each bar-group in a chart represents the results for expressions of a specified size. Sizes range from 10 to 2560, quadrupling for each successive sample. Bars in each group represent the effects of standardisation, fusion and lifting. We do not include in these charts bars representing the results of pressing and catalogue look-ups: for random inputs, test results indicate that these transformations have little effect after lifting has done its work.

Even for random inputs excluding 0 and 1 as sub-expressions, standardisation reduces size by 10%–20%. In comparison, the linear-time transformations from [5] are slightly less effective, by 1%–5%.

Fusion achieves further reductions by 5%–20% of the original size. The effect is greater for small $\Sigma$ because there is a higher likelihood of common-factor prefixes or suffixes in choices.

Lifting has the greatest effect, and its effectiveness increases for larger input expressions. This is because in larger expressions generated at random it is more likely that sub-expressions specify or include total languages over a sub-alphabet, for which the simplifying rules of lifting apply. Again, size-reductions are greatest for small $\Sigma$: in the binary case, even the smallest inputs are almost halved in size (the mean is 52.48% for inputs of size 10), and the largest are reduced to a tiny fraction (the mean is 0.37% for inputs of size 2560). As expected, the gains from lifting diminish as alphabet size increases, yet for $\#\Sigma = 8$ we still see further gains of 20% or more for larger inputs.

Looking at the actual outputs of the lifting transformation, one often encounters the minimal expression for the total language, e.g. $(a + b)^*$ for $\#\Sigma = 2$. The lifting rules that produce such total expressions only depend on context-free properties: for any fixed alphabet $\Sigma$ there is a context-free grammar that describes the expressions transformed by lifting into $\Sigma^*$. Instead of relying on random samples to observe their frequency, we can count how many expressions of a given size satisfy a given context-free property [11]. For $\#\Sigma = n$ each expression $x$ falls into one of $2 \cdot n + 4$ classes: two for each possible size of $\alpha_1(x)$ ($\mathrm{ewp}(x)$ and $\neg\mathrm{ewp}(x)$), plus two special classes for $\Sigma^+$ and $\Sigma^*$. This final class contains exactly the expressions transformed by lifting into the minimal expression for the total language of alphabet size $n$. The proportion

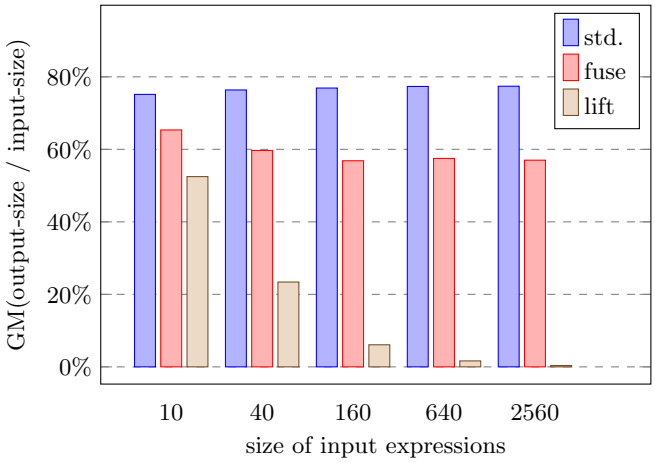


Figure 1: Output size as a percentage of input size, for samples of 1000 randomly-generated inputs of each size ($\#\Sigma = 2$).

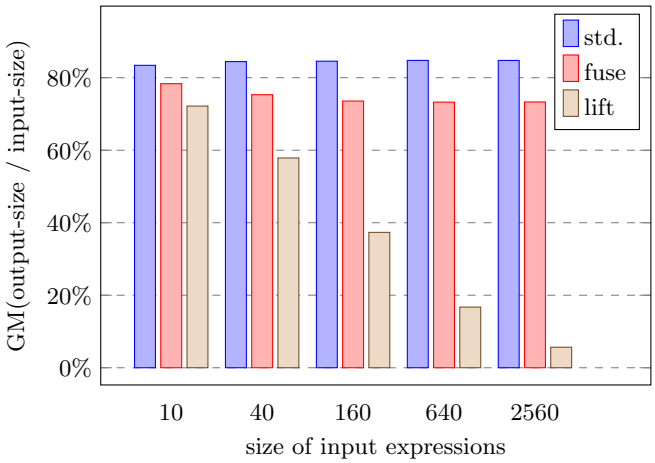


Figure 2: Output size as a percentage of input size, for samples of 1000 randomly-generated inputs of each size ($\#\Sigma = 4$).

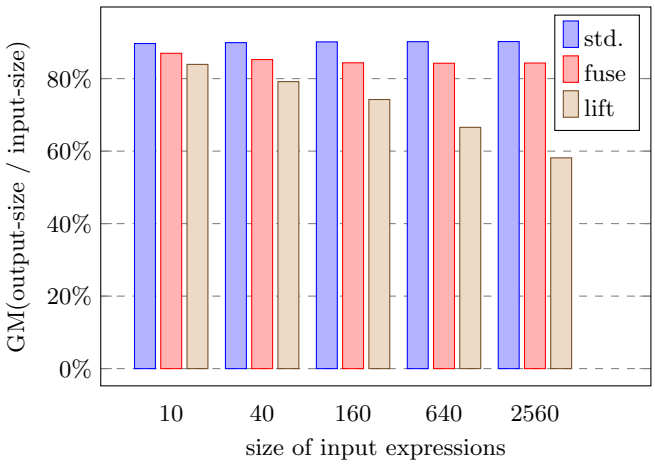


Figure 3: Output size as a percentage of input size, for samples of 1000 randomly-generated inputs of each size ($\#\Sigma = 8$).

of expressions belonging to this class grows with expression size. It converges to a limit, and it converges faster for small alphabet sizes.

In relation to Figures 1–3, for expression size 2560 and for $\#\Sigma = 2, 4, 8$ the proportions are 42.8%, 18.0% and 5.1%. So, indeed 42.8% of *all* expressions of size 2560 over the alphabet $\{a, b\}$ will be transformed by lifting into $(a + b)^*$ (size 4), which goes some way to explain why the mean result size on our sample is as low as $0.0037 \cdot 2560 \approx 9.5$.

These observations independently confirm a recently published result drawn to our attention by a reviewer. Uniform random expressions include with high probability large subexpressions that collapse under an "absorbing pattern" of simplification, and the expected size of random expressions after such simplification is constant [12].

### *5.2. Effectiveness for Systematically Derived Expressions*

Testing using expressions generated at random in a specified variety of sizes is arguably "fair", and taking the ratio between input and output sizes as a measure of simplification is arguably "natural". However, both choices limit the kinds of insights that can be obtained from test results. Expressions generated at random are not typical of those arising in most applications, where expressions are used to describe the characteristics of well-defined systems. And even the most detailed analysis of output sizes relative to input sizes tells us nothing about how close output expressions are to the minimum possible size.

Another consideration, despite the observation about lifting for small alphabets, is that we expect expressions generated at random to be artificially difficult to simplify. Generally, random assemblies resist compression, even by techniques exploiting deep knowledge of what assemblies mean.

For all these reasons, we also test our simplifying transformations using as inputs expressions that are systematically constructed, and have known minimal equivalents. We call this construction *HU-expansion* as it complicates an expression by first translating into a non-deterministic FSA (using the technique from [13]), and then using a highly expansive back-translation into expressions, inspired by a proof-of-concept method taken from [2].

The method iteratively creates expressions $R_{ij}$ connecting any two states $i, j$ of the FSA. Starting with the FSA's transitions only, subsequent iterations of $R_{ij}$ are obtained by a technique similar to matrix multiplication. The process is repeated until maximum path length is reached. The final result is given by the sum of all $R_{sf}$ for which $s$ is initial and $f$ is final.

The original algorithm in [2] adds to each $R_{ij}$ in the $k$-th iteration the connections via state $k$, that is: $R_{ik} \cdot (R_{kk})^* \cdot R_{kj}$. Instead, we add in the $k$-th iteration the connections from $i$ to $j$ via *all* states: $\sum_{p \in S} R_{ip} \cdot (R_{pp})^* \cdot R_{pj}$. This modification means that the round-trip can increase the size of expressions drastically. Expressions of size $n$

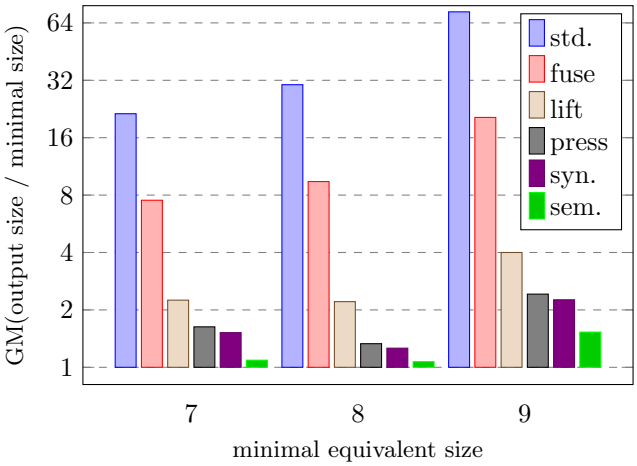


Figure 4: Output size as a multiple of minimal-equivalent size, for exhaustive test inputs recursively derived from automata ($\#\Sigma = 2$).

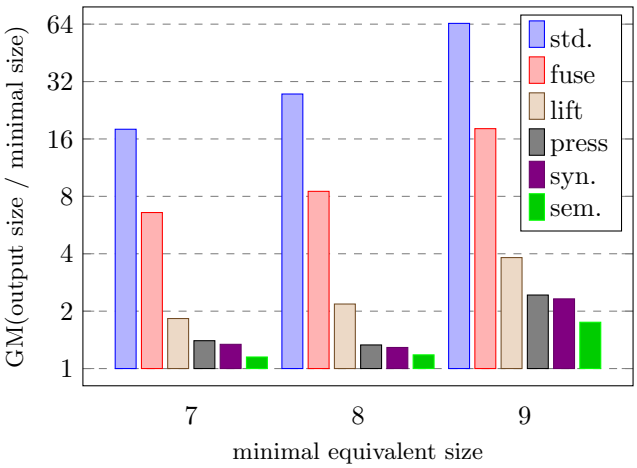


Figure 5: Output size as a multiple of minimal-equivalent size, for exhaustive test inputs recursively derived from automata ($\#\Sigma = 3$).

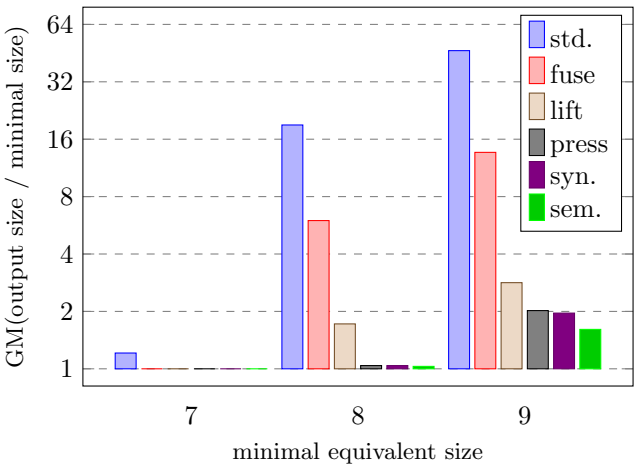


Figure 6: Output size as a multiple of minimal-equivalent size, for exhaustive test inputs recursively derived from automata ($\#\Sigma = 4$).

have an alphabetic width (number of alphabetic symbols) of at most $w = \lceil n/2 \rceil$, and Antimirov's construction creates FSAs with at most $w + 1$ states. In the worst cases, we blow up the size of expressions from 7, 8 and 9 to 7528, 52395 and 424761, respectively, even though none of the FSAs has more than 6 states.

Such large expressions are obtained even though we apply standardisation throughout. They pose a suitable challenge for our simplifying transformations.

In summary, for each of the 11,680 catalogued minimal expressions of size $7 \ldots 9$ and with $\#\Sigma = 2 \ldots 4$ we construct a corresponding non-deterministic FSA that is subsequently HU-expanded to obtain a test expression already in standardised form. This collection of expressions is systematically complete within the inevitable bounds of computing resources: it covers every regular language specifiable by an expression of up to size 9.

Figures 4–6 present the results for all six transformations (standardisation, fusion, lifting, pressing, syntactic look-up and semantic look-up) applied to all 11,680 HU-expanded expressions. For each transformation and minimal size category we compute the geometric mean ratio of output size to minimal size, giving an effectiveness measure for a complete class of tests.

The size reductions obtained by adding successive transformations are so much greater than they were for random inputs, that we plot the mean size ratios on a logarithmic scale. For example, whereas Figure 1 shows that on average fusion reduces the size of standardised random expressions over a binary alphabet by 10%–20%, Figure 4 shows that for standardised HU-constructed expressions it reduces size by a factor of three or four, or by around 66%–75%.

Even the final-stage transformation, using look-up in the semantic catalogue, does not in all cases discover a minimal form of HU-expanded expressions even though these languages are represented in the catalogue. This is a consequence of the size-bounding restrictions explained in §4.4: look-up in the semantic catalogue is only attempted for expressions beneath a certain size.

In Figure 6, the bar group for size 7 represents exceptionally small multiples of the minimal size. The reason is that the very few expressions $x$ with $\text{size}(x) = 7$ and $|\alpha(x)| = 4$ contain neither stars nor queries.

### *5.3. Run-time Cost*

We need a simple measure of the cost of a simplifying transformation applied to a test-set of input expressions. We choose the average run-time to simplify a single test expression.

All timings given in this section are for our implementation in Haskell, compiled using ghc version 8.6.5 for execution on a 3GHz iMac computer.

Figures 7–9 show average run-times in milliseconds plotted on a log-scale ranging from a microsecond to a tenth of a second. Plots are given for the full range of transformations described in §3: standardisation, fusion, lifting,

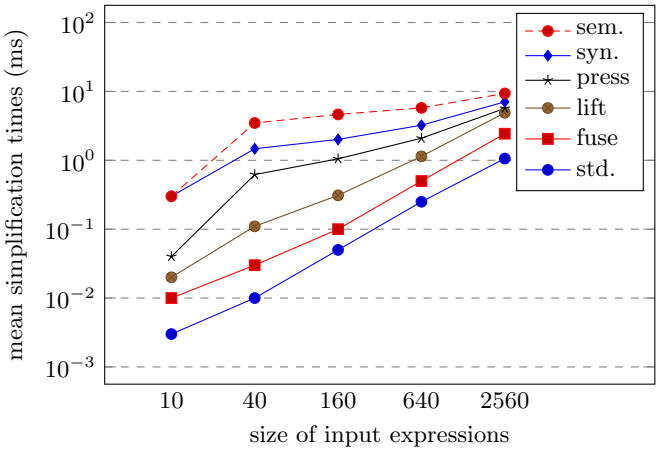


Figure 7: Mean simplification times for random input expressions ($\#\Sigma = 2$).

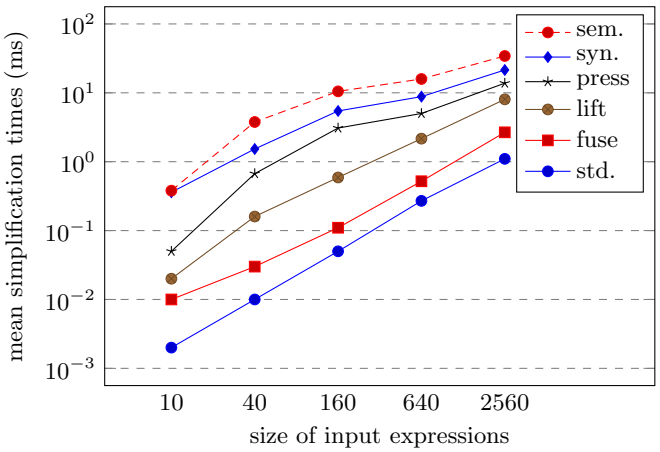


Figure 8: Mean simplification times for random input expressions ($\#\Sigma = 4$).

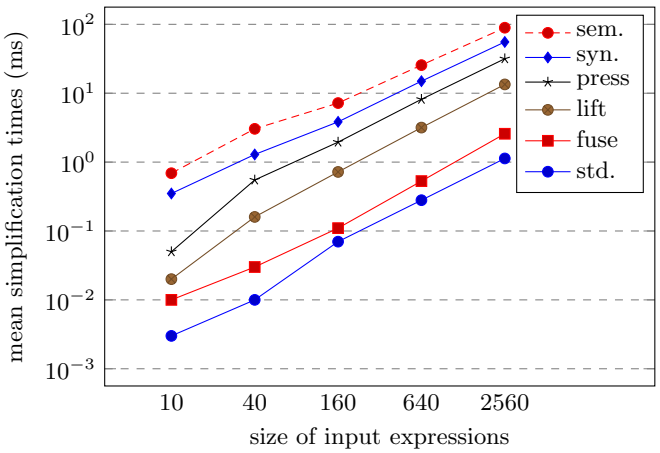


Figure 9: Mean simplification times for random input expressions ($\#\Sigma = 8$).

pressing, syntactic catalogue look-up & semantic catalogue look-up. The test-sets of input expressions are the same samples of randomly generated expressions as were used for Figures 1–3.

For standardisation and fusion the data is consistent with their expected log-linear runtime. The few parts of lifting that require quadratic effort are barely detectable in the data: lifting still shows log-linear growth on random inputs. Runtimes for pressing show a bump before size-boundedness sets in. All transformations beyond lifting have the advantage of starting with lifted intermediate expressions, so they rarely operate directly on expressions of a similar size to the original inputs.

For Figure 9, recall from §4 that our catalogues only contain data for alphabet sizes up to 4. So catalogue-based transformations are only applicable for sub-expressions with at most 4 distinct symbols. All expressions of size 10 satisfy that constraint automatically. Alphabet size also affects the distribution: random expressions of size 10 for $\#\Sigma = 8$ contain fewer repeated symbols, and fewer stars and queries, than random expressions for $\#\Sigma = 4$.

## 6. Discussion and Conclusions

Typically, simplification methods reported in published papers such as [4, 5, 14] apply a small selection of algebraic equivalences. Using only equivalences between terms without repeated variables avoids comparison of expressions and allows simplification in linear time. For example, the Gruber-Gulan transformation [5] follows this approach. For reasons of space, we did not include this transformation in our results section, but its effect on random expressions is very modest: for all populations we tested, the mean output size is greater than for any of our own methods.

In contrast, our approach has been to find various ways of exploiting comparisons. Even standardisation, our most basic transformation, uses a basic structural comparison of expressions to eliminate duplicate choices. Successive transformations use more sophisticated comparisons up to full-language equivalence and orderings.

Inevitably, our transformations are more costly than the linear-time simplifications from [5]. However, by careful representation choices and memoisation, many of our comparisons can be made quite cheap. All the alphabetic expression attributes we need can be computed in linear time, assuming a constant-sized alphabet. Our test results show that in practice overall costs of transformation are no worse than log-linear — until we start using full language comparisons.

Lifting is extremely effective on random expressions, especially over small alphabets where the average size of a lifted normal form seems eventually constant — regardless of input size! Indeed, when simplifying random expressions, after the lifting stage most of the size-reduction has already been achieved. Even for HU-expanded inputs our test results show a marked benefit from lifting, albeit less dramatic than for random inputs. The benefits of the later stage transformations are far more apparent in the results for HU-expansions, and these we claim are more typical for systematically derived expressions.

In closing, what summary advice would we give to developers of tools and applications involving regular expressions, by way of practical recommendation? How can they best make use of the various transformations we have set out?

Unless there is some compelling reason to store unprocessed expressions in full (e.g. in an educational setting) at least keep them in standardised form. If processing of expressions is one of the main computational tasks, the potentially significant size-reductions of fusion and lifting are attractive. The modest additional cost of these transformations over linear-bounded algebraic simplification will typically be offset by reduced application-specific expression processing. When an overall expression result is to be presented to a user, or provided as a primary ingredient in output, consider the option of applying the more costly transformations involving full language comparisons. Pressing and catalogue look-up give expressions within a very small factor of minimal size; at greater cost, they can be made even more effective by raising their various internal size-limits.


## Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors. Much of the work was done when the second author was a visiting professor at the University of Kent, funded by the University of York.


## Github Repository

See the repository at `github.com/ColinRunciman/MrE` for the implementation of our simplifier and various auxiliary programs used to obtain our test results. A README file includes instructions for reproducing measured results from which we extracted data for the charts and graphs in §5. Clearly, run-times (§5.3) may vary depending on the implementation platform. Some effectiveness figures (§5.1) may vary a little depending on the sample populations randomly generated by an installation, but with high probability they will be very close to those we have presented.

To aid maintaining correctness during development, the repository includes a program that searches for counter-examples using Braquehais' *LeanCheck* library [15]. The program checks for semantic counter-examples, where the transformed expression has not the same language as the original, or syntactic counter-examples where the size of the expression increases.

## References


[1] S. C. Kleene, Representation of Events in Nerve Nets and Finite Automata, Princeton University Press, 1951, pp. 3–42.

[2] J. E. Hopcroft, J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.

[3] L. Clemente, R. Mayr, Efficient reduction of nondeterministic automata with application to language inclusion testing, Log. Methods Comput. Sci. 15 (1) (2019). doi:10.23638/LMCS-15(1:12)2019.
URL `https://doi.org/10.23638/LMCS-15(1:12)2019`

[4] A. Salomaa, Two complete axiom systems for the algebra of regular events, J. ACM 13 (1) (1966) 158–169.

[5] H. Gruber, S. Gulan, Simplifying regular expressions : A quantitative perspective, in: Languages and Automata Theory and Applications (LATA 2010), Springer LNCS 6031, 2010, pp. 285–296.

[6] H. Hunt, On the time and tape complexity of languages I, in: STOC'73 Proceedings of the fifth annual ACM Symposium on the Theory of Computing, 1973, pp. 10–19.

[7] Haskell: an advanced, purely functional programming language, last accessed September 2020 (2020).
URL `www.haskell.org`

[8] J. A. Brzozowski, Derivatives of regular expressions, JACM 11 (1964) 481—-494.

[9] M. Almeida, Equivalence of regular languages: an algorithmic approach and complexity analysis, Ph.D. thesis, University of Porto (2010).

[10] W. Gutjahr, Uniform random generation of expressions respecting algebraic identities, Computing 47 (1991) 51–67.

[11] T. Hickey, J. Cohen, Uniform generation of strings in a context-free language, SIAM Journal of Computing 12 (4) (1983) 645–655.

[12] F. Koechlin, C. Nicaud, P. Rotondo, Uniform Random Expressions Lack Expressivity, in: P. Rossmanith, P. Heggernes, J.-P. Katoen (Eds.), 44th Intl. Symposium on Mathematical Foundations of Computer Science (MFCS 2019), Vol. 138 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 51:1–51:14.

[13] V. Antimirov, Partial derivatives of regular expressions and finite automaton constructions, Theoretical Computer Science 155 (2) (1996) 291–319. doi:https://doi.org/10.1016/0304-3975(95)00182-4.

[14] A. A. Trejo Ortiz, G. Fernández Anaya, Regular expression simplification, Mathematics and Computers in Simulation 45 (1) (1998) 59 – 71. doi:https://doi.org/10.1016/S0378-4754(97)00086-4.

[15] R. M. Braquehais, Tools for discovery, refinement and generalization of functional properties by enumerative testing, Ph.D. thesis, University of York (2017).

[16] J. H. Conway, Regular Algebra and Finite Machines, Chapman and Hall, 1971.


## Appendix A. Proofs for illustrative fusion rules

We illustrate here the correctness of the transformation rules for fusion, excluding laws that are axioms for Kleene-algebras. We assume standardisation laws. We also use the C11 law from [16]: $(x+y)^* = (x^* \cdot y)^* \cdot x^*$.

**Fusion Law 1.** $(x + y^*)^* = (x + y)^*$

*Proof.*

$$\begin{array}{ll} & (x + y^*)^* \\ = & \{\text{standardisation}\} \\ & (y^* + x)^* \\ = & \{C11\} \\ & ((y^*)^* \cdot x)^* \cdot (y^*)^* \\ = & \{\text{standardisation}\} \\ & (y^* \cdot x)^* \cdot y^* \\ = & \{C11\} \\ & (y + x)^* \\ = & \{\text{standardisation}\} \\ & (x + y)^* \end{array}$$

□

**Fusion Law 2.** $(x + y?)^* = (x + y)^*$

*Proof.*

$$\begin{array}{ll} & (x + y?)^* \\ = & \{\text{standardisation}\} \\ & (y? + x)^* \\ = & \{C11\} \\ & (y?^* \cdot x)^* \cdot y?^* \\ = & \{\text{standardisation}\} \\ & (y^* \cdot x)^* \cdot y^* \\ = & \{C11\} \\ & (y + x)^* \\ = & \{\text{standardisation}\} \\ & (x + y)^* \end{array}$$

□

**Fusion Law 3.** $x^* \cdot x^* = x^*$

*Proof.*

$$\begin{array}{ll} & x^* \cdot x^* \\ = & \{\text{standardisation}\} \\ & (x^* \cdot 1)^* \cdot x^* \\ = & \{C11\} \\ & (x + 1)^* \\ = & \{\text{standardisation}\} \\ & x^* \end{array}$$

□

## Appendix B. Proofs for illustrative pressing rules

We illustrate here the correctness of the transformation rules for pressing that involve the $\searrow$ operator on expressions. Semantically, correctness requires that:

$$L((x \searrow y) + y) = L(x + y)$$

Syntactically, a useful transformation can only proceed if in addition $\text{size}(x \searrow y) < \text{size}(x)$, but that is not our concern here. In the following, "$x = y$" between expressions refers to their semantic equality $L(x) = L(y)$.

We shall prove the correctness of each of the following illustrative rules from §3.4.

$$x? \searrow y = x \quad \textbf{if} \quad \text{ewp}(y) \tag{B.1}$$

$$x \cdot y \searrow z^* = x \cdot (y \searrow z^*) \quad \textbf{if} \quad L(x) \subseteq L(z^*) \tag{B.2}$$

**Pressing Law 1.** $(x \searrow y) + y = x + y$

We proof this by structural induction on $x$, splitting the proof into the cases for B.1 and B.2.

*Proof.* For the B.1 case we have:

$$\begin{array}{ll} & (x? \searrow y) + y \\ = & \{\text{rule B.1}\} \\ & x + y \\ = & \{\text{ewp}(y) \text{ from B.1, } y = y?\} \\ & x + y? \\ = & \{\text{standardisation}\} \\ & x? + y \end{array}$$

□

For B.2 the following lemma is useful.

**Lemma 1.** *If* $L(x) \subseteq L(z^*)$ *then* $z^* = x \cdot z^* + z^*$

*Proof.* If $L(x) \subseteq L(z^*)$ then $x + z^* = z^*$, and so:

$$\begin{array}{ll} & z^* \\ = & \{\text{fusion}\} \\ & z^* \cdot z^* \\ = & \{\ x + z^* = z^*\} \\ & (x + z^*) \cdot z^* \\ = & \{\text{distributivity}\} \\ & x \cdot z^* + z^* \cdot z^* \\ = & \{\text{fusion}\} \\ & x \cdot z^* + z^* \end{array}$$

□

Now we resume the proof of the pressing law.

*Proof.* Case B.2:

$$\begin{array}{ll} & ((x \cdot y) \searrow z^*) + z^* \\ = & \{\text{rule B.2}\} \\ & (x \cdot (y \searrow z^*)) + z^* \\ = & \{\text{Lemma 1}\} \\ & (x \cdot (y \searrow z^*)) + x \cdot z^* + z^* \\ = & \{\text{distributivity}\} \\ & x \cdot ((y \searrow z^*) + z^*) + z^* \\ = & \{\text{induction hypothesis}\} \\ & x \cdot (y + z^*) + z^* \\ = & \{\text{distributivity}\} \\ & x \cdot y + x \cdot z^* + z^* \\ = & \{\text{Lemma 1}\} \\ & x \cdot y + z^* \end{array}$$

□