Self-certifying Railroad Diagrams

Or: How to Teach Nondeterministic Finite Automata

Ralf Hinze(✉)

Technische Universität Kaiserslautern, 67653 Kaiserslautern, Germany
ralf-hinze@cs.uni-kl.de

Abstract. Regular expressions can be visualized using railroad or syntax diagrams. The construction does not depend on fancy artistic skills. Rather, a diagram can be systematically constructed through simple, local transformations due to Manna. We argue that the result can be seen as a nondeterministic finite automaton with ϵ-transitions. Despite its simplicity, the construction has a number of pleasing characteristics: the number of states and the number of edges is linear in the size of the regular expression; due to sharing of sub-automata and auto-merging of states the resulting automaton is often surprisingly small. The proof of correctness relies on the notion of a subfactor. In fact, Antimirov's subfactors (partial derivatives) appear as target states of non-ϵ-transitions, suggesting a smooth path to nondeterministic finite automata without ϵ-transitions. Antimirov's subfactors, in turn, provide a fine-grained analysis of Brzozowski's factors (derivatives), suggesting a smooth path to deterministic finite automata. We believe that this makes a good story line for introducing regular expressions and automata.

1 Introduction

Everything should be as simple as it can be, but not simpler.

Albert Einstein

Regular expressions and finite automata have a long and interesting history, see Fig. 1. Kleene introduced regular expressions in the 1950's to represent events in nerve nets [10]. In the early 60's, Brzozowski [6] presented an elegant method for translating a regular expression to a deterministic finite automaton (DFA). His algorithm is based on two general properties of languages, sets of words. A language can be partitioned into a set of non-empty words and the remainder (ϵ is the set containing the empty word, + is set union).

$$L = (L - \epsilon) + (L \cap \epsilon)$$

If L is given by a regular expression, the remainder $L \cap \epsilon$ can be readily computed (is L "nullable"?). A non-nullable language can be further factored:

$$L - \epsilon = a_1 \cdot (a_1 \backslash L) + \cdots + a_n \cdot (a_n \backslash L) \tag{1}$$

G. Hutton (Ed.): MPC 2019, LNCS 11825, pp. 103–137, 2019.
https://doi.org/10.1007/978-3-030-33636-3_5

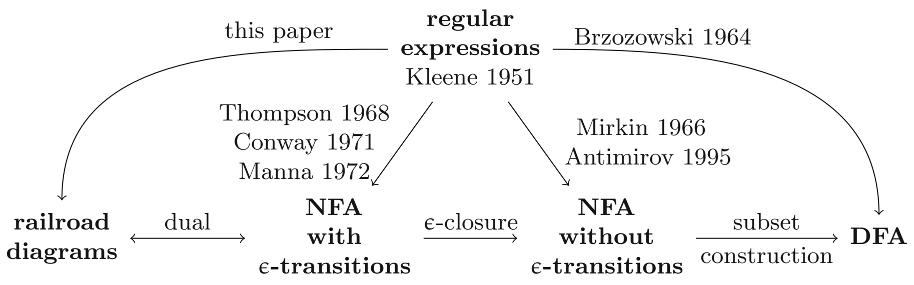


Fig. 1. A brief history of regular expressions and finite automata.

where $a_1, \ldots, a_n$ are *distinct* symbols of the alphabet and $a_1 \backslash L, \ldots, a_n \backslash L$ are so-called *right factors* of L, see Sect. 3. The right factor of a regular language is again a regular language; iterating the two steps above yields a DFA. However, some care has to be exercised when implementing the algorithm: while there is only a finite number of *semantically* different right factors, they have infinitely many syntactic representations.

Thirty years later, Antimirov [3] ameliorated the problem, providing a fine-grained analysis of Brzozowski's approach. The central idea is to loosen the requirement that the symbols in (1) are distinct. In his linear form

$$L - \epsilon = a_1 \cdot L_1 + \cdots + a_n \cdot L_n \tag{2}$$

the $a_1, \ldots, a_n$ are arbitrary symbols of the alphabet, *not necessarily distinct.* Each of the languages L_i is a *right subfactor* of the left-hand side: $L - \epsilon \supseteq a_i \cdot L_i$. Because the requirement of distinctness is dropped, Antimirov's approach yields a nondeterministic finite automaton without ϵ-transitions. (Brzozowski's automaton can be recovered via the subset construction, basically grouping the subfactors by symbol.) Antimirov's representation of subfactors ensures that there is only a finite number of *syntactically* different right subfactors. Unfortunately, the argument is still somewhat involved.

An alternative advocated in this paper is to approach the problem from the other side, see Fig. 1. Regular expressions can be visualized using railroad or syntax diagrams. (I have first encountered syntax diagrams in Jensen and Wirth's "Pascal: User Manual and Report" [9].) Adapting a construction due to Manna [12], we show that a diagram can be systematically constructed through simple, local transformations. We argue that the result can be seen as a nondeterministic finite automaton with ϵ-transitions. Moreover, the automaton contains all Antimirov subfactors as target states of non-ϵ-transitions.

The remainder of the paper is organized in two strands, which are only loosely coupled:

– Sections 2, 4, and 6 introduce railroad diagrams and prove Manna's construction correct. The material is targeted at first-year students; an attempt is made to explain the construction in basic terms without, however, compromising on precision and concision. Central to the undertaking is the consistent use of inequalities and reasoning using Galois connections.

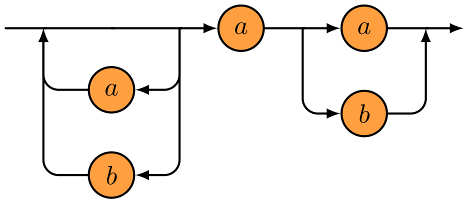
– Sections 3 and 5 highlight the theoretical background, which is based on regular algebra. The "student material" is revised providing shorter, but slightly more advanced accounts. Section 7 links railroad diagrams to Antimirov's subfactors.

The paper makes the following contributions:

– We suggest a story line for introducing regular expressions and automata, using a visual representation of regular expressions as the starting point.
– We show that diagrams form a regular algebra and we identify a sub-algebra, self-certifying diagrams, which supports simple correctness proofs.
– We show that Manna's construction can be implemented in eight lines of Haskell and highlight some of its salient features: sharing of sub-automata and auto-merging of states.
– We prove that a Manna automaton contains the subfactors of an Antimirov automaton as target states of non-ϵ-transitions.

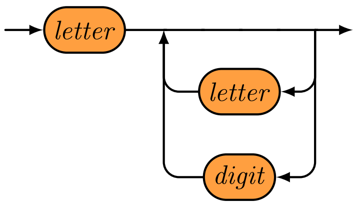
2 Railroad Diagrams

Regular expressions can be nicely visualized using so-called railroad or syntax diagrams. Consider the regular expression $(a \mid b)^* \cdot a \cdot (a \mid b)$ over the alphabet $\Sigma = \{a, b\}$, which captures the language of all words whose penultimate symbol is an a. Its railroad diagram is shown below.



The diagram has one entry on the left, the starting point, and one exit on the right, the rail destination. Each train journey from the starting point to the destination generates a word of the language, obtained by concatenating the symbols encountered on the trip.

In the example above, the "stations" contain symbols of the alphabet. We also admit arbitrary languages (or representations of languages) as labels, so that we can visualize a regular expression or, more generally, a language at an appropriate level of detail. In the diagram below, which captures the lexical syntax of identifiers in some programming language, we have chosen not to expand the languages of letters and digits.



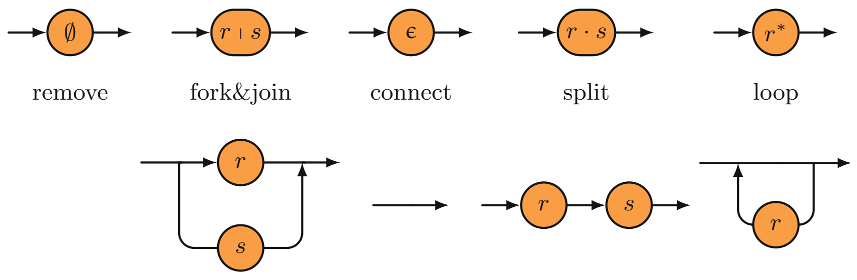
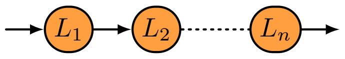


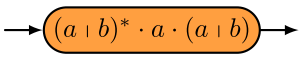
Fig. 2. Manna's rules for constructing a railroad diagram (the diagrams in the first row are translated into the corresponding diagrams in the second row).

In this more general setting, a single train journey visiting the stations

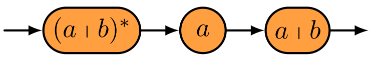


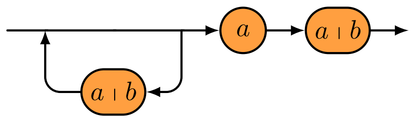
generates the language $L_1 \cdot L_2 \cdot \ldots \cdot L_n$.

The generalization to languages has the added benefit that we can construct a railroad diagram for a given regular expression in a piecemeal fashion using simple *local* transformations. The point of departure is a diagram that consists of a single station, containing the given expression. To illustrate, for our running example we have:



Then we apply the transformations shown in Fig. 2 until we are happy with the result. Each rule replaces a single station by a smallish railroad diagram that generates the same language. Observe that there is no rule for a single symbol—stations containing a single symbol cannot be refined. We refer to the transformation rules as *Manna's construction*. The diagrams below show two intermediate stages of the construction for our running example.

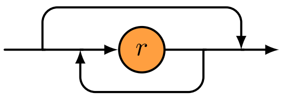


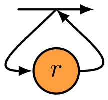


The transformations emphasize the close correspondence between regular expressions and railroad diagrams: composition is visualized by chaining two tracks; choice by forking and joining a track; iteration by constructing a loop. Each rule in Fig. 2 is compositional, for example, a station for $r \mid s$ is transformed into a diagram composed of one station for r and a second station for s. Quite attractively, the rules guarantee that the resulting diagram is planar so it can be easily drawn without crossing tracks.

When drawing diagrams by hand we typically exercise some artistic licence. For example, there are various ways to draw loops. The diagram for r^* in Fig. 2 has the undesirable feature that the sub-diagram for r is drawn from right to left.

Alternative drawings that avoid this unfortunate change of direction include:





Railroad diagrams are visually appealing, but there is more to them. If we apply the transformations of Fig. 2 until all stations either contain ϵ or a single symbol, then we obtain an NFA, a nondeterministic finite automaton with ϵ-transitions. Before we make this observation precise in Sect. 4, we first introduce some background.

Exercise 1. Devise diagrams for non-zero repetitions $r^+ = r \cdot r^*$ and for optional occurrences $\epsilon \mid r$.

3 Interlude: Regular Algebras and Regular Expressions

This section details the syntax and semantics of regular expressions. It is heavily based on Backhouse's axiomatization of regular algebras [4], which, while equivalent to Conway's "Standard Kleene Algebra" or **S**-algebra [8], stresses the importance of factors.[1] I suggest that you skip the section on first reading, except perhaps for notation.

Regular Algebras. A regular algebra $(R, \leqslant, \sum, 1, \cdot)$ is a blend of a complete lattice and a monoid, interfaced by two Galois connections:

1. $(R, \leqslant)$ is a complete lattice with join $\sum$,
2. $(R, 1, \cdot)$ is a monoid,
3. for all $p \in R$, the partially applied functions $(p\,\cdot\,)$ and $(\,\cdot\,p)$ are both left adjoints in Galois connections between $(R, \leqslant)$ and itself.

We use $0 = \sum \emptyset$ and $a_1 + a_2 = \sum\{a_1, a_2\}$ as short cuts for the least element (nullary join) and binary joins. The right adjoints of $(p\,\cdot\,)$ and $(\,\cdot\,p)$ are written $(p \setminus\,)$ and $(\,/\,p)$.

Calculationally speaking, it is most useful to capture the first and the third requirement as equivalences. Join or least upper bound is defined by the following equivalence, which incidentally also establishes a Galois connection.

$$\sum A \leqslant b \iff \forall a \in A \;.\; a \leqslant b \tag{3a}$$

Instantiated to nullary and binary joins, (3a) specializes to $0 \leqslant b \iff \mathit{true}$ (that is, 0 is indeed the least element) and $a_1 + a_2 \leqslant b \iff a_1 \leqslant b \wedge a_2 \leqslant b$.

[1] Actually, Conway's axiomatization is incomplete as pointed out by Abramsky and Vickers [1]: only if the axiom $\sum\{a\} = a$ is added, his **S**-algebras are equivalent to regular algebras.

The interface between the lattice and the monoid structure is given by the following two equivalences.

$$p \cdot a \leqslant b \quad \iff \quad a \leqslant p \setminus b \tag{3b}$$

$$a \cdot p \leqslant b \quad \iff \quad a \leqslant b / p \tag{3c}$$

The element $p \setminus b$ is called a *right factor of* b (or left quotient or left derivative).

The axioms have a wealth of consequences. Adjoint functions are order-preserving; left adjoints preserve joins; right adjoints preserve meets. Consequently, the axioms imply:

$$\sum A \cdot b = \sum \{\, a \cdot b \mid a \in A \,\} \tag{4a}$$

$$a \cdot \sum B = \sum \{\, a \cdot b \mid b \in B \,\} \tag{4b}$$

Instantiated to nullary and binary joins, property (4a) specializes to $0 \cdot b = 0$ and $(a_1 + a_2) \cdot b = (a_1 \cdot b) + (a_2 \cdot b)$. So, composition distributes over choice.

Truth values ordered by implication form a regular algebra: $(\mathbb{B}, \Rightarrow, \exists, \mathit{true}, \wedge)$, where $\exists\, B = (\mathit{true} \in B)$. Note that the right adjoint of $(p \wedge\ )$ is $(p \Rightarrow\ )$.

Languages, sets of words over some alphabet Σ, are probably the most prominent example of a regular algebra. The example is actually an instance of a more general construction. Let $(R, 1, \cdot)$ be some monoid. We can lift the monoid to a monoid on sets, setting $1 = \{1\}$ and $A \cdot B = \{\, a \cdot b \mid a \in A, b \in B \,\}$. (Note that here and elsewhere we happily overload symbols: in $1 = \{1\}$, the first occurrence of 1 denotes the unit of the lifted monoid, whereas the second occurrence is the unit of the underlying monoid.) It remains to show that $(P \cdot\ )$ and $(\ \cdot P)$ are left adjoints. We show the former; the calculation for the latter proceeds completely analogously.

$$\begin{aligned}
& P \cdot A \subseteq B \\
\iff & \quad \{ \text{ definition of composition } \} \\
& \{\, p \cdot a \mid p \in P, a \in A \,\} \subseteq B \\
\iff & \quad \{ \text{ set inclusion } \} \\
& \forall p \in R, a \in R \,.\, p \in P \wedge a \in A \Longrightarrow p \cdot a \in B \\
\iff & \quad \{ \ (x \wedge\ ) \text{ is a left adjoint, } (x \Rightarrow\ ) \text{ preserves universal quantification } \} \\
& \forall a \in R \,.\, a \in A \Longrightarrow (\forall p \in R \,.\, p \in P \Longrightarrow p \cdot a \in B) \\
\iff & \quad \{ \text{ set inclusion } \} \\
& A \subseteq \{\, a \in R \mid \forall p \in R \,.\, p \in P \Longrightarrow p \cdot a \in B \,\}
\end{aligned}$$

The right adjoint $P \setminus B$ is given by the formula on the right. Consequently, $(\mathcal{P}(R), \subseteq, \bigcup, 1, \cdot)$ is a regular algebra. In the case of languages, the underlying monoid is the free monoid $(\Sigma^*, \epsilon, \cdot)$, where ϵ is the empty word and composition is concatenation of words.

Occasionally, it is useful to make stronger assumptions, for example, to require that the underlying lattice is Boolean. In this case, $(\ + p)$ has a left adjoint, which we write $(\ - p)$.

Iteration. The axioms of regular algebra explicitly introduce choice and composition. Iteration is a derived concept: a^* is defined as the least solution of the inequality $1 + a \cdot x \leqslant x$ in the unknown x (which is guaranteed to exist because of completeness).

$$1 + a \cdot a^* \leqslant a^* \tag{5a}$$

$$\forall x \,.\, a^* \leqslant x \iff 1 + a \cdot x \leqslant x \tag{5b}$$

Property (5a) states that a^* is indeed a solution; the so-called fixed-point induction principle (5b) captures that a^* is the least among all solutions. The two formulas have a number of important consequences,

$$1 \leqslant a^* \qquad a^* \cdot a^* \leqslant a^* \qquad a \leqslant a^* \qquad (a^*)^* = a^*$$

which suggest why a^* is sometimes called the reflexive, transitive closure of a.

Exercise 2. Show that iteration is order-preserving: $a \leqslant b \Longrightarrow a^* \leqslant b^*$.

Exercise 3. Assuming a Boolean lattice, show $a^* = (a - 1)^*$.

Regular Expressions. Regular expressions introduce syntax for choice, composition, and iteration. Expressed as a Haskell datatype, they read:

```
type Alphabet = Char
data Reg
  = Empty             -- the empty language
  | Alt  Reg Reg      -- choice
  | Eps               -- the empty word
  | Sym Alphabet      -- a single symbol
  | Cat  Reg Reg      -- composition
  | Rep  Reg          -- iteration
```

We use the term *basic symbol* to refer to the empty word or a single symbol of the alphabet.

A regular expression denotes a language. However, there is only one constructor specific to languages: *Sym*, which turns an element of the underlying alphabet into a regular expression. The other constructors can be interpreted generically in any regular algebra.

$$\begin{array}{llll}
[\![\, \mathit{Empty} \,]\!] & = [\![\, \emptyset \,]\!] & = 0 \\
[\![\, \mathit{Alt}\ r\ s \,]\!] & = [\![\, r \mid s \,]\!] & = [\![\, r \,]\!] + [\![\, s \,]\!] \\
[\![\, \mathit{Eps} \,]\!] & = [\![\, \epsilon \,]\!] & = 1 \\
[\![\, \mathit{Sym}\ a \,]\!] & = [\![\, a \,]\!] & = \{a\} \\
[\![\, \mathit{Cat}\ r\ s \,]\!] & = [\![\, r \cdot s \,]\!] & = [\![\, r \,]\!] \cdot [\![\, s \,]\!] \\
[\![\, \mathit{Rep}\ r \,]\!] & = [\![\, r^* \,]\!] & = [\![\, r \,]\!]^*
\end{array}$$

The second column of the semantic equations introduces alternative notation for the Haskell constructors. They serve the sole purpose of improving the readability of examples. (Occasionally, we also omit the semantic brackets, mixing syntax and semantics.)

We additionally introduce "smart" versions of the constructors that incorporate basic algebraic identities.

```
cat :: Reg → Reg → Reg      alt :: Reg → Reg → Reg     rep :: Reg → Reg
cat Empty s = Empty         alt Empty s = s            rep Empty = Eps
cat r Empty = Empty         alt r Empty = r            rep Eps   = Eps
cat Eps s   = s             alt r s     = Alt r s      rep r     = Rep r
cat r Eps   = r
cat r s     = Cat r s
```

They ensure, in particular, that *Empty* never occurs as a sub-expression. More sophisticated identities such as $(r^*)^* = r^*$ are not captured, however, to guarantee constant running time.

Regular Homomorphisms. Whenever a new class of structures is introduced, the definition of structure-preserving maps follows hard on its heels. Regular algebras are no exception. A regular homomorphism is a monoid homomorphism that preserves joins.[2] (Or equivalently, a monoid homomorphism that is a left adjoint). For example, the test whether a language contains the empty word ("is nullable") is a regular homomorphism from languages $(\mathcal{P}(\Sigma^*), \subseteq, \bigcup, \{\epsilon\}, \cdot)$ to Booleans $(\mathbb{B}, \Rightarrow, \exists, \mathit{true}, \wedge)$. Since membership $(x \in\ )$ is a left adjoint, it remains to check that $(\epsilon \in\ )$ preserves composition and its unit,

$$\epsilon \in \{\epsilon\} = \mathit{true}$$
$$\epsilon \in A \cdot B = \epsilon \in A \ \wedge\ \epsilon \in B$$

which is indeed the case.

The nullability check can be readily implemented for regular expressions.

```
nullable :: Reg → Bool
nullable (Empty)   = False
nullable (Alt r s) = nullable r ∨ nullable s
nullable (Eps)     = True
nullable (Sym a)   = False
nullable (Cat r s) = nullable r ∧ nullable s
nullable (Rep r)   = True
```

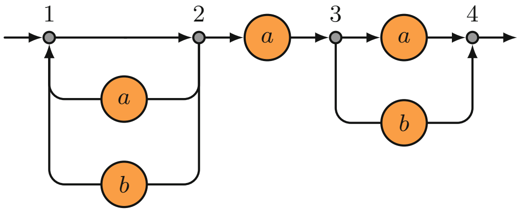
The call *nullable r* yields *True* if and only if $\epsilon \in [\![r]\!]$.

Exercise 4. Let $(R, \leqslant, \sum, 1, \cdot)$ be a regular algebra. Prove that $\sum : \mathcal{P}(R) \to R$ is a regular homomorphism from the algebra of lifted monoids to R.

[2] A regular algebra is really a blend of a *complete join-semilattice* and a monoid. It is a standard result of lattice theory that a complete join-semilattice is a complete lattice. However, a complete join-semilattice homomorphism is not necessarily a complete lattice homomorphism, as there is no guarantee that it also preserves meets.

4 Finite Automata with ϵ-Transitions

To be able to give railroad diagrams a formal treatment, we make fork and join points of tracks explicit. Our running example, the diagram for $(a \mid b)^* \cdot a \cdot (a \mid b)$, consists of six tracks, where a track is given by a source point, a labelled station, and a target point, see Fig. 3. In the example, points are natural numbers; labels are either the empty word, omitted in the pictorial representation, or symbols of the underlying alphabet.



$$\{(1, \epsilon, 2), (2, a, 1), (2, b, 1), (2, a, 3), (3, a, 4), (3, b, 4)\}$$

Fig. 3. Railroad diagram for $(a \mid b)^* \cdot a \cdot (a \mid b)$.

In general, the points of a diagram are drawn from some fixed set V; its labels are elements of some monoid $(R, 1, \cdot)$. A diagram G is a set of arrows: $G \subseteq V \times R \times V$. For the application at hand, labels are given by languages: $R := \mathcal{P}(\Sigma^*)$ for some fixed alphabet Σ. An arrow (q, a, z) is sometimes written $a : q \to z$ for clarity.

Diagrams as Generators. A diagram generates a language for each pair of points q and z: we concatenate the labels along each path from q to z; the union of these languages is the language generated, in symbols $q \circ\!\!-\!\!\circ z$.

The diagram in Fig. 3 is now called $1 \circ\!\!-\!\!\circ 4$ (from entry to exit), and we have just said that it represents our running example, so we now want to formally prove that $1 \circ\!\!-\!\!\circ 4 = (a \mid b)^* \cdot a \cdot (a \mid b)$—and by the way, the sub-diagrams also equal sub-languages, so, for example, we claim that $1 \circ\!\!-\!\!\circ 2 = (a \mid b)^*$.

A path is a finite, possibly empty sequence of arrows with matching endpoints. To reason about languages generated by a diagram, we make use of the following three properties.

$$1 \subseteq i \circ\!\!-\!\!\circ i \tag{6a}$$

$$L \subseteq i \circ\!\!-\!\!\circ j \quad \Longleftarrow \quad (i, L, j) \in G \tag{6b}$$

$$(i \circ\!\!-\!\!\circ j) \cdot (j \circ\!\!-\!\!\circ k) \subseteq i \circ\!\!-\!\!\circ k \tag{6c}$$

Properties (6a) and (6c) imply that we can go around in a loop a finite number of times.

$$(i \circ\!\!-\!\!\circ i)^* \subseteq i \circ\!\!-\!\!\circ i \tag{6d}$$

The proof is a straightforward application of the fixed-point induction principle.

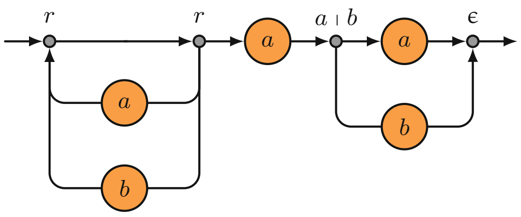
$$\begin{aligned}
& (i \circ\!\!-\!\!\circ i)^* \subseteq i \circ\!\!-\!\!\circ i \\
\Longleftarrow \quad & \{\text{ fixed-point induction (5b) }\} \\
& 1 + (i \circ\!\!-\!\!\circ i) \cdot (i \circ\!\!-\!\!\circ i) \subseteq i \circ\!\!-\!\!\circ i \\
\Longleftrightarrow \quad & \{\text{ join } (3a) \} \\
& 1 \subseteq i \circ\!\!-\!\!\circ i \quad \wedge \quad (i \circ\!\!-\!\!\circ i) \cdot (i \circ\!\!-\!\!\circ i) \subseteq i \circ\!\!-\!\!\circ i
\end{aligned}$$

Using these inequalities we can, for example, show that the language generated by the diagram in Fig. 3 contains at least $(a \mid b)^* \cdot a \cdot (a \mid b)$.

$$\begin{aligned}
& (a \mid b)^* \cdot a \cdot (a \mid b) \\
= \quad & \{\text{ unit of composition }\} \\
& \epsilon \cdot ((a \mid b) \cdot \epsilon)^* \cdot a \cdot (a \mid b) \\
\subseteq \quad & \{\text{ arrows (6b) and monotonicity of operators }\} \\
& (1 \circ\!\!-\!\!\circ 2) \cdot ((2 \circ\!\!-\!\!\circ 1) \cdot (1 \circ\!\!-\!\!\circ 2))^* \cdot (2 \circ\!\!-\!\!\circ 3) \cdot (3 \circ\!\!-\!\!\circ 4) \\
\subseteq \quad & \{\text{ composition (6c) and iteration (6d) }\} \\
& (1 \circ\!\!-\!\!\circ 2) \cdot (2 \circ\!\!-\!\!\circ 2) \cdot (2 \circ\!\!-\!\!\circ 3) \cdot (3 \circ\!\!-\!\!\circ 4) \\
\subseteq \quad & \{\text{ composition (6c) }\} \\
& 1 \circ\!\!-\!\!\circ 4
\end{aligned}$$

Of course, we would actually like to show that $1 \circ\!\!-\!\!\circ 4$ is equal to $(a \mid b)^* \cdot a \cdot (a \mid b)$. We could argue that there are no other paths that contribute to the language generated. While this is certainly true, the statement does not seem to lend itself to a nice calculational argument. Fortunately, there is an attractive alternative, which we dub self-certifying diagrams.

Self-certifying Diagrams. The central idea is to record our expectations about the languages generated in the diagram itself, using languages as points: $V := \mathcal{P}(\Sigma^*)$. Continuing our running example, we replace the natural numbers of Fig. 3 by languages, see Fig. 4. The arrow $(r, a, a \mid b)$, for example, records that we can generate the language r if we start at the source point of the arrow; if we start at its end point, we can only generate a or b. In general, a point is identified with the language of all paths from the position to the unique exit. (As an aside, note that the point r is drawn twice in Fig. 4. Our mathematical model does not distinguish between these two visual copies. So the arrow (r, ϵ, r) is actually a self-loop, which could be safely removed.)



$\{(r, \epsilon, r), (r, a, r), (r, b, r), (r, a, a \mid b), (a \mid b, a, \epsilon), (a \mid b, b, \epsilon)\}$

Fig. 4. Annotated railroad diagram for $r = (a \mid b)^* \cdot a \cdot (a \mid b)$.

Of course, not every combination of languages makes sense. We require that the target is a *subfactor* of the source:

$$\begin{tikzcd} \alpha \circ \arrow[r] & \textcircled{L} \arrow[r] & \circ\, \beta \end{tikzcd} \quad \text{is admissible} \quad :\iff \quad \alpha \supseteq L \cdot \beta \tag{7}$$

Observe the use of inequalities: we require $\alpha \supseteq L \cdot \beta$, not $\alpha = L \cdot \beta$. Equality is too strong as there may be further arrows with the same source. The source point only provides an upper bound on the language generated from this point onward.

Theorem 1. *If all arrows of a diagram are admissible, then*

$$\alpha \supseteq (\alpha \circ\!\!-\!\!\circ \beta) \cdot \beta \tag{8}$$

for all points $\alpha \in V$ *and* $\beta \in V$.

Returning briefly to our running example, it is not too hard to check that all arrows in Fig. 4 are admissible. We can therefore conclude that $r \supseteq (r \circ\!\!-\!\!\circ \epsilon) \cdot \epsilon$. Since we already know that the diagram generates at least r, we have $r \circ\!\!-\!\!\circ \epsilon = r$, as desired.

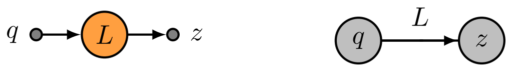
Proof (Theorem 1). First, we show that a single path is admissible if all of its arrows are. The empty path is admissible as $\alpha \supseteq \epsilon \cdot \alpha$. The concatenation of two admissible paths again gives an admissible path:

$$\begin{aligned}
& \alpha \supseteq K \cdot \beta \quad \wedge \quad \beta \supseteq L \cdot \gamma \\
\Longrightarrow \quad & \{ \text{ monotonicity of composition } \} \\
& \alpha \supseteq K \cdot (L \cdot \gamma) \\
\Longleftrightarrow \quad & \{ \text{ associativity of composition } \} \\
& \alpha \supseteq (K \cdot L) \cdot \gamma
\end{aligned}$$

The language $Q \circ\!\!-\!\!\circ Z$ is the join of all languages generated by paths from Q to Z. To establish the theorem it suffices to show that admissibility is closed under arbitrary joins.

$$\begin{aligned}
& \forall L \in \mathcal{L} \;.\; \alpha \supseteq L \cdot \beta \\
\iff & \quad \{ \text{ join (3a) } \} \\
& \alpha \supseteq \bigcup \{ L \cdot \beta \mid L \in \mathcal{L} \} \\
\iff & \quad \{ \text{ composition distributes over join (4a) } \} \\
& \alpha \supseteq (\bigcup \mathcal{L}) \cdot \beta
\end{aligned}$$

Diagrams as Acceptors. So far we have emphasized the generative nature of diagrams. However, if a diagram is simple enough, it can also be seen as an acceptor. Indeed, if all the stations contain a basic symbol, then the diagram amounts to an NFA. The difference is merely one of terminology and presentation. In automata theory, points are called states and arrows are called transitions. Furthermore, transitions are typically drawn as labelled edges, as on the right below.



(I prefer railroad diagrams over standard drawings of automata, as the latter seem to put the visual emphasis on the wrong entity but, perhaps, this is a matter of personal taste.)

You may want to skim through the next section on first reading and proceed swiftly to Sect. 6, which justifies Manna's construction.

5 Interlude: The Regular Algebra of Diagrams

The purpose of this section is to investigate the algebraic structure of diagrams. We show that diagrams also form a regular algebra, provided the labels are drawn from a monoid, and we identify an important sub-algebra: self-certifying diagrams.

If R is a monoid, then diagrams $(\mathcal{P}(V \times R \times V), \subseteq, \bigcup, 1, \cdot)$ form a regular algebra where

$$1 = \{ (i, 1, i) \mid i \in V \} \tag{9a}$$

$$F \cdot G = \{ (i, a \cdot b, k) \mid \exists j \in V \;.\; (i, a, j) \in F, (j, b, k) \in G \} \tag{9b}$$

The unit is the diagram of self-loops $1 : i \to i$; composition takes an arrow $a : i \to j$ in F and an arrow $b : j \to k$ in G to form an arrow $a \cdot b : i \to k$ in $F \cdot G$. Diagrams are a blend of lifted monoids and relations: for $V := \mathbf{1}$ we obtain the regular algebra of lifted monoids; for $R := \mathbf{1}$ we obtain the regular algebra of (untyped) relations. (Here $\mathbf{1}$ is a singleton set.)

The lattice underlying diagrams is a powerset lattice and hence complete. It is not too hard to show that composition is associative with 1 as its unit. It remains to prove that $(P\cdot)$ and $(\cdot P)$ are left adjoints. As before, we only provide one calculation.

$$\begin{array}{ll}
 & P \cdot A \subseteq B \\
\iff & \{ \text{ definition of composition (9b) } \} \\
 & \{\, (i, p \cdot a, k) \mid \exists j \in V \,.\, (i,p,j) \in P, (j,a,k) \in A \,\} \subseteq B \\
\iff & \{ \text{ set inclusion } \} \\
 & \forall i \in V, p \in R, a \in R, k \in V \,. \\
 & \qquad (\exists j \in V \,.\, (i,p,j) \in P \wedge (j,a,k) \in A) \Longrightarrow (i, p \cdot a, k) \in B \\
\iff & \{ \text{ existential quantification is the join in the lattice of predicates } \} \\
 & \forall i \in V, p \in R, j \in V, a \in R, k \in V \,. \\
 & \qquad (i,p,j) \in P \wedge (j,a,k) \in A \Longrightarrow (i, p \cdot a, k) \in B \\
\iff & \{ \text{ } (x \wedge\ ) \text{ is a left adjoint } \} \\
 & \forall i \in V, p \in R, j \in V, a \in R, k \in V \,. \\
 & \qquad (j,a,k) \in A \Longrightarrow ((i,p,j) \in P \Longrightarrow (i, p \cdot a, k) \in B) \\
\iff & \{ \text{ } (x \Rightarrow\ ) \text{ preserves universal quantification } \} \\
 & \forall j \in V, a \in R, k \in V \,. \\
 & \qquad (j,a,k) \in A \Longrightarrow (\forall i \in V, p \in R \,.\, (i,p,j) \in P \Longrightarrow (i, p \cdot a, k) \in B) \\
\iff & \{ \text{ set inclusion } \} \\
 & A \subseteq \{\, (j,a,k) \mid \forall i \in V, p \in R \,.\, (i,p,j) \in P \Longrightarrow (i, p \cdot a, k) \in B \,\}
\end{array}$$

The right adjoint $P \setminus B$ is given by the formula on the right.

For the remainder of the section we assume that R is a regular algebra. The translation of regular expressions to diagrams reduces the word problem (is a given word w an element of the language denoted by the regular expression r?) to a path-finding problem (is there a w-labelled path in the diagram corresponding to r?). Now, a directed path in the diagram G is an arrow in G^*, the reflexive, transitive closure of G. Using the closure of a diagram, we can provide a more manageable definition of $q \circ\!\!-\!\!\circ z$, the language generated by *paths* from q to z in G: we set $q \circ\!\!-\!\!\circ z := q \circ\!\!-\!(G^*)\!-\!\!\circ z$ where

$$q \circ\!\!-\!(F)\!-\!\!\circ z = \sum \{\, a \in R \mid (q,a,z) \in F \,\} \tag{10}$$

is the language generated by all *arrows* from q to z in F.

It is appealing that we can use regular algebra to reason about the semantics of regular expressions, their visualization using diagrams, as well as their "implementation" in terms of nondeterministic finite automata. For example, in Sect. 4 we have mentioned in passing that we can safely remove ϵ-labelled self-loops from a diagram. This simple optimization is justified by $G^* = (G - 1)^*$, a general property of iteration, see Exercise 3.

Exercise 5. Let $D = \{(0, \epsilon, 1), (0, L, 0)\}$. Calculate D^* using the so-called *star decomposition* rule $(F + G)^* = F^* \cdot G \cdot F^* \cdot G \cdot \ldots \cdot G \cdot F^* = F^* \cdot (G \cdot F^*)^*$.

Admissibility Revisited. In Sect. 4 we have introduced the concept of an admissible arrow. (Section 6, which proves Manna's construction correct, relies heavily on this concept.) Let us call a diagram admissible if all its arrows are admissible. An alternative, but equivalent definition builds on $\multimapinv(G)\multimap$:

$$G \text{ admissible} \quad :\Longleftrightarrow \quad \forall \alpha \in V, \beta \in V \,.\, \alpha \supseteq (\alpha \multimapinv(G)\multimap \beta) \cdot \beta$$

The proof that the two definitions are indeed equivalent is straightforward.

$$\begin{aligned}
& \forall \alpha \in V, \beta \in V \,.\, \alpha \supseteq (\alpha \multimapinv(G)\multimap \beta) \cdot \beta \\
\Longleftrightarrow \quad & \{ \text{ definition of } \multimapinv(G)\multimap \ (10) \ \} \\
& \forall \alpha \in V, \beta \in V \,.\, \alpha \supseteq \Big(\sum \{ L \mid (\alpha, L, \beta) \in G \} \Big) \cdot \beta \\
\Longleftrightarrow \quad & \{ \text{ composition distributes over join (4a) } \} \\
& \forall \alpha \in V, \beta \in V \,.\, \alpha \supseteq \sum \{ L \cdot \beta \mid (\alpha, L, \beta) \in G \} \\
\Longleftrightarrow \quad & \{ \text{ join (3a) } \} \\
& \forall (\alpha, L, \beta) \in G \,.\, \alpha \supseteq L \cdot \beta
\end{aligned}$$

Admissible (or self-certifying) diagrams form a sub-algebra of the algebra of diagrams $(\mathcal{P}(V \times R \times V), \subseteq, \bigcup, 1, \cdot)$: the unit 1 is admissible, $F \cdot G$ is admissible if both F and G are, $\sum \mathcal{G}$ is admissible if every diagram $G \in \mathcal{G}$ is. (The proof of Theorem 1 shows exactly that.) As a consequence,

$$G^* \text{ admissible} \quad \Longleftarrow \quad G \text{ admissible}$$

which is the import of Theorem 1.

Mathematical Models of Diagrams. There are many options for modelling diagrams or graphs. One important consideration is whether edges have an identity. A common definition of a labelled graph introduces two sets, a set of nodes and a set of edges with mappings from edges to source node, label, and target node. While our definition admits multiple directed edges between two nodes, it cannot capture multiple edges with the same label, a feature that we do not consider important for the application at hand. Backhouse's notion of a matrix [4] simplifies further by not allowing multiple edges at all. In his model a graph[3] is a function $V \times V \to R$ that maps a pair of nodes, source and target, to an element of the underlying regular algebra.[4] Matrices ordered pointwise also form a regular algebra, provided R is one.

[3] Backhouse uses the terms matrix and graph interchangeably.

[4] Backhouse's definition is actually more general: a matrix is given by a function $r \to R$ where $r \subseteq V \times V$ is a fixed relation, the dimension of the matrix. This allows him to distinguish between non-existent edges and edges that are labelled with 0.

We haven chosen to allow multiple edges as we wish to model Manna's transformation for choice, which replaces a single edge by two edges with the same source and target, see Fig. 2. In some sense, the operation $q \circ\!\!-\!\!(G)\!\!-\!\!\circ\, z$ undoes this step by joining the labels of all arrows from q to z. In fact, the operation is a regular homomorphism from diagrams to matrices, mapping a diagram G to a matrix $\circ\!\!-\!\!(G)\!\!-\!\!\circ$, where application to source and target points is written $q \circ\!\!-\!\!(G)\!\!-\!\!\circ\, z$. We need to establish the following three properties (the right-hand sides of the formulas implicitly define join, unit, and composition of matrices).

$$q \circ\!\!-\!\!(\bigcup \mathcal{G})\!\!-\!\!\circ\, z = \sum \{\, q \circ\!\!-\!\!(G)\!\!-\!\!\circ\, z \mid G \in \mathcal{G} \,\} \tag{11a}$$

$$q \circ\!\!-\!\!(1)\!\!-\!\!\circ\, z = (1 \lhd q = z \rhd 0) \tag{11b}$$

$$q \circ\!\!-\!\!(F \cdot G)\!\!-\!\!\circ\, z = \sum \{\, (q \circ\!\!-\!\!(F)\!\!-\!\!\circ\, i) \cdot (i \circ\!\!-\!\!(G)\!\!-\!\!\circ\, z) \mid i \in V \,\} \tag{11c}$$

Here $a \lhd c \rhd b$ is shorthand for **if** c **then** a **else** b, known as the Hoare conditional choice operator. The proof of (11a) relies on properties of set comprehensions.

$$\begin{aligned}
& q \circ\!\!-\!\!(\bigcup \mathcal{G})\!\!-\!\!\circ\, z \\
= \quad & \{ \text{ definition (10) } \} \\
& \sum \left\{ a \in R \mid (q, a, z) \in \bigcup \mathcal{G} \right\} \\
= \quad & \{ \text{ set comprehension } \} \\
& \sum \{\, a \in R \mid G \in \mathcal{G}, (q, a, z) \in G \,\} \\
= \quad & \{ \text{ book-keeping law, see below } \} \\
& \sum \left\{ \sum \{\, a \in R \mid (q, a, z) \in G \,\} \mid G \in \mathcal{G} \right\} \\
= \quad & \{ \text{ definition (10) } \} \\
& \sum \{\, q \circ\!\!-\!\!(G)\!\!-\!\!\circ\, z \mid G \in \mathcal{G} \,\}
\end{aligned}$$

The penultimate step of the proof uses the identity $\sum \{ \sum \{ a_i \mid i \in I \} \mid I \in \mathcal{I} \} = \sum \{ a_i \mid I \in \mathcal{I}, i \in I \}$. Written in a point-free style the formula is known as the book-keeping law: $\sum \circ \mathcal{P} \sum = \sum \circ \bigcup$. (Categorically speaking, the identity follows from the fact that every complete join-semilattice is an algebra for the powerset monad, see Ex. VI.2.1 in [11].)

For (11b) we reason,

$$
\begin{aligned}
& q \mathrel{\circ\!\!-}(1)\mathrel{-\!\!\circ} z \\
= \quad & \{ \text{ definition (10) } \} \\
& \textstyle\sum \{ a \in R \mid (q, a, z) \in 1 \} \\
= \quad & \{ \text{ definition of unit (9a) } \} \\
& \textstyle\sum \{ a \in R \mid (q, a, z) \in \{ (i, 1, i) \mid i \in V \} \} \\
= \quad & \{ \text{ membership } \} \\
& \textstyle\sum \{ 1 \mid q = z \} \\
= \quad & \{ \text{ set comprehension } \} \\
& 1 \lhd q = z \rhd 0
\end{aligned}
$$

The proof of (11c) relies again on the book-keeping law. The calculation is most perspicuous if read from bottom to top.

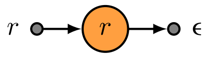
$$
\begin{aligned}
& q \mathrel{\circ\!\!-}(F \cdot G)\mathrel{-\!\!\circ} z \\
= \quad & \{ \text{ definition (10) } \} \\
& \textstyle\sum \{ x \in R \mid (q, x, z) \in F \cdot G \} \\
= \quad & \{ \text{ definition of composition (9b) } \} \\
& \textstyle\sum \{ x \in R \mid (q, x, z) \in \{ (q, a \cdot b, z) \mid \exists i \in V \;.\; (q, a, i) \in F, (i, b, z) \in G \} \} \\
= \quad & \{ \text{ membership } \} \\
& \textstyle\sum \{ a \cdot b \mid \exists i \in V \;.\; (q, a, i) \in F, (i, b, z) \in G \} \\
= \quad & \{ \text{ book-keeping law, see above } \} \\
& \textstyle\sum \Big\{ \sum \{ a \cdot b \mid (q, a, i) \in F, (i, b, z) \in G \} \mid i \in V \Big\} \\
= \quad & \{ \text{ composition preserves joins (4a) } \} \\
& \textstyle\sum \Big\{ \Big(\sum \{ a \in R \mid (q, a, i) \in F \}\Big) \cdot \Big(\sum \{ b \in R \mid (i, b, z) \in G \}\Big) \mid i \in V \Big\} \\
= \quad & \{ \text{ definition (10) } \} \\
& \textstyle\sum \{ (q \mathrel{\circ\!\!-}(F)\mathrel{-\!\!\circ} i) \cdot (i \mathrel{\circ\!\!-}(G)\mathrel{-\!\!\circ} z) \mid i \in V \}
\end{aligned}
$$

Exercise 6. Show that properties (6a)–(6c) are consequences of (11a)–(11c).

6 Construction of Finite Automata with ϵ-Transitions

We now have the necessary prerequisites in place to formalize and justify Manna's construction of diagrams. Recall that the construction works by repeatedly replacing a single arrow by a small diagram until all labels are basic symbols.

The point of departure is the diagram below that consists of a single, admissible arrow,

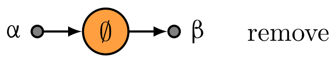
G_0 :


where r is the given regular expression. Clearly, we have $r \circ\!\!-\!(G_0^*)\!-\!\!\circ\, \epsilon = r$, where $q \circ\!\!-\!(F)\!-\!\!\circ\, z$ is the language generated by all arrows from q to z in F (10).

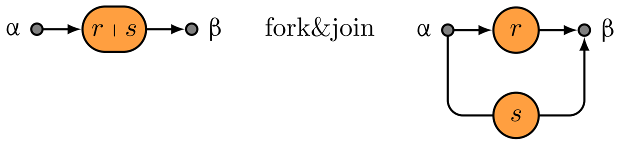
For each of the following transformations we show that (1) admissibility of arrows is preserved (*correctness*); and (2) for each path in the original diagram, there is a corresponding path in the transformed diagram (*completeness*). Correctness implies that for each graph G_i generated in the process we have $r \supseteq r \circ\!\!-\!(G_i^*)\!-\!\!\circ\, \epsilon$. Completeness ensures that $Q \circ\!\!-\!(G_i^*)\!-\!\!\circ\, Z \subseteq Q \circ\!\!-\!(G_{i+1}^*)\!-\!\!\circ\, Z$, which implies $r \subseteq r \circ\!\!-\!(G_i^*)\!-\!\!\circ\, \epsilon$.[5]

We consider each of the transformations of Fig. 2 in turn.

Case $\emptyset$: there is nothing to prove.



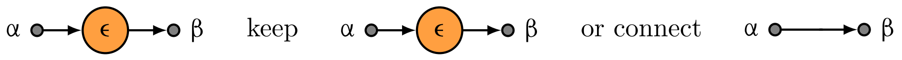
Case $r \mid s$: we replace the single arrow by two arrows sharing source and target.



To establish correctness and completeness, we reason:

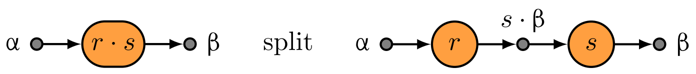
$$\begin{aligned}
& \alpha \supseteq (r \mid s) \cdot \beta \\
\iff\; & \{\ \text{distributivity (4a)}\ \} \\
& \alpha \supseteq (r \cdot \beta) \mid (s \cdot \beta) \\
\iff\; & \{\ \text{join (3a)}\ \} \\
& \alpha \supseteq r \cdot \beta \quad \wedge \quad \alpha \supseteq s \cdot \beta
\end{aligned}
\qquad\qquad
\begin{aligned}
& r \mid s \\
\subseteq\; & \{\ \text{arrows (6b) and monotonicity}\ \} \\
& (\alpha \circ\!\!-\!\!\!-\!\!\circ\, \beta) \cup (\alpha \circ\!\!-\!\!\!-\!\!\circ\, \beta) \\
=\; & \{\ \text{idempotency}\ \} \\
& \alpha \circ\!\!-\!\!\!-\!\!\circ\, \beta
\end{aligned}$$

Case ϵ: we keep ϵ-arrows (we may choose to omit ϵ-labels in diagrams though),



so there is nothing to prove.

Case $r \cdot s$: we split the composition introducing an intermediate point.



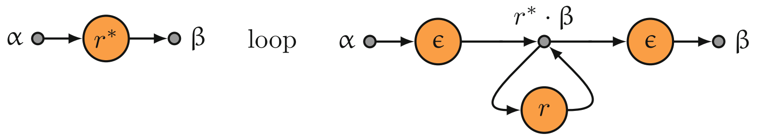
[5] We use "correctness" and "completeness" only in this very narrow sense.

We have to show that the two arrows on the right are admissible and that they generate $r \cdot s$.

$$\begin{aligned}
& \alpha \supseteq (r \cdot s) \cdot \beta \\
\iff & \quad \{ \text{ associativity } \} \\
& \alpha \supseteq r \cdot (s \cdot \beta) \\
\iff & \quad \{ \text{ reflexivity } \} \\
& \alpha \supseteq r \cdot (s \cdot \beta) \quad \wedge \quad s \cdot \beta \supseteq s \cdot \beta
\end{aligned}$$

$$\begin{aligned}
& r \cdot s \\
\subseteq & \quad \{ \text{ arrows (6b) and monotonicity } \} \\
& (\alpha \mathrel{\circ\!\!-\!\!\circ} s \cdot \beta) \cdot (s \cdot \beta \mathrel{\circ\!\!-\!\!\circ} \beta) \\
\subseteq & \quad \{ \text{ composition (6c) } \} \\
& \alpha \mathrel{\circ\!\!-\!\!\circ} \beta
\end{aligned}$$

Case r^*: we introduce an intermediate point, $i = r^* \cdot \beta$, which serves as source and target of a looping arrow.



We have to show that the three arrows are admissible and that they generate r^*. The arrow $\alpha \mathrel{\circ\!\!-\!(\epsilon)\!-\!\!\circ} r^* \cdot \beta$ is trivially admissible since $\alpha \mathrel{\circ\!\!-\!(r^*)\!-\!\!\circ} \beta$ is. It remains to show

$$\begin{aligned}
& i \supseteq i \\
\iff & \quad \{ \; r^* = \epsilon \mid r \cdot r^* \; \} \\
& i \supseteq (\epsilon \mid r \cdot r^*) \cdot \beta \\
\iff & \quad \{ \text{ distributivity (4a) } \} \\
& i \supseteq (\epsilon \cdot \beta) \mid (r \cdot i) \\
\iff & \quad \{ \text{ join (3a) } \} \\
& i \supseteq \epsilon \cdot \beta \quad \wedge \quad i \supseteq r \cdot i
\end{aligned}$$

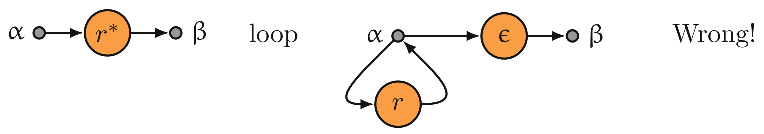
$$\begin{aligned}
& \epsilon \cdot r^* \cdot \epsilon \\
\subseteq & \quad \{ \text{ arrows (6b) and monotonicity } \} \\
& (\alpha \mathrel{\circ\!\!-\!\!\circ} i) \cdot (i \mathrel{\circ\!\!-\!\!\circ} i)^* \cdot (i \mathrel{\circ\!\!-\!\!\circ} \beta) \\
\subseteq & \quad \{ \text{ iteration (6d) } \} \\
& (\alpha \mathrel{\circ\!\!-\!\!\circ} i) \cdot (i \mathrel{\circ\!\!-\!\!\circ} i) \cdot (i \mathrel{\circ\!\!-\!\!\circ} \beta) \\
\subseteq & \quad \{ \text{ composition (6c) } \} \\
& \alpha \mathrel{\circ\!\!-\!\!\circ} \beta
\end{aligned}$$

A few remarks are in order.

The calculations are entirely straightforward. The calculations on the left rely on basic properties of regular algebra; the calculations on the right capture visual arguments—it is quite obvious that for each path in the original diagram there is a corresponding path in the transformed diagram. That's the point—the calculations should be simple as the material is targeted at first-year students.

Quite interestingly, we need not make any assumptions about disjointness of states, which is central to the McNaughton-Yamada-Thompson algorithm [2]. For example, there is no guarantee that the intermediate state for composition, $s \cdot \beta$, is not used elsewhere in the diagram. Admissibility of arrows ensures that sharing of sub-diagrams is benign. We will get back to this point shortly.

It is perhaps tempting to leave out the intermediate point i for iteration:



While the diagram on the right is complete, it is not correct: the arrow (α, r, α) is not admissible, consider, for example, $\alpha = a \mid b^*$ and $r = b^*$. Since the arrow (α, r, α) is generally part of a larger diagram, it might contribute to other edges emerging from α. A simple case analysis shows that the intermediate point i is necessary—there is no diagram involving only α and β that will do the trick.

Exercise 7. Do the transformations work in both directions?

Implementation in Haskell. Manna's transformation rules can be seen as the specification of a non-deterministic algorithm as the rules can be applied in any order. However, they serve equally well as the basis for an inductive construction. We represent arrows by triples.

```
type Arrow label = (Reg, label, Reg)
diagram :: Reg → Set (Arrow (Basic Alphabet))
diagram r = diagram′ (r, r, Eps)
```

The worker function *diagram′* maps a single arrow to a diagram, a set of arrows. (We assume the existence of a suitable library for manipulating finite sets.) Basic symbols are represented by elements of *Basic Alphabet*: *Eps′* represents the empty word ϵ, *Sym′ a* represents a singleton word, the symbol *a*.

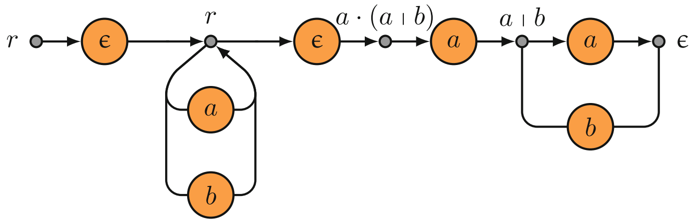
```
data Basic a = Eps′ | Sym′ a
diagram′ :: Arrow Reg → Set (Arrow (Basic Alphabet))
diagram′ (α, Empty, β)  = ∅
diagram′ (α, Alt r s, β)  = diagram′ (α, r, β) ∪ diagram′ (α, s, β)
diagram′ (α, Eps, β)      = {(α, Eps′, β)}
diagram′ (α, Sym a, β)  = {(α, Sym′ a, β)}
diagram′ (α, Cat r s, β) = diagram′ (α, r, i) ∪ diagram′ (i, s, β)
  where i = cat s β
diagram′ (α, Rep r, β)   = {(α, Eps′, i)} ∪ diagram′ (i, r, i) ∪ {(i, Eps′, β)}
  where i = cat (Rep r) β
```

Voilà. A regular expression compiler in eight lines.

The following session shows the algorithm in action (*a* is shorthand for *Sym* 'a', *b* for *Sym* 'b', and *ab* for *Alt a b*).

$$
\begin{array}{l}
\ggg \quad \mathit{diagram}\ (\mathit{Cat}\ (\mathit{Rep}\ ab)\ (\mathit{Cat}\ a\ ab)) \\
\{(\mathit{Cat}\ (\mathit{Rep}\ ab)\ (\mathit{Cat}\ a\ ab), \mathit{Eps}', \mathit{Cat}\ (\mathit{Rep}\ ab)\ (\mathit{Cat}\ a\ ab)) \\
\ \ (\mathit{Cat}\ (\mathit{Rep}\ ab)\ (\mathit{Cat}\ a\ ab), \mathit{Sym}'\ \texttt{'a'}, \mathit{Cat}\ (\mathit{Rep}\ ab)\ (\mathit{Cat}\ a\ ab)) \\
\ \ (\mathit{Cat}\ (\mathit{Rep}\ ab)\ (\mathit{Cat}\ a\ ab), \mathit{Sym}'\ \texttt{'b'}, \mathit{Cat}\ (\mathit{Rep}\ ab)\ (\mathit{Cat}\ a\ ab)) \\
\ \ (\mathit{Cat}\ (\mathit{Rep}\ ab)\ (\mathit{Cat}\ a\ ab), \mathit{Eps}', \mathit{Cat}\ a\ ab) \\
\ \ (\mathit{Cat}\ a\ ab, \mathit{Sym}'\ \texttt{'a'}, ab) \\
\ \ (ab, \mathit{Sym}'\ \texttt{'a'}, \mathit{Eps}), \\
\ \ (ab, \mathit{Sym}'\ \texttt{'b'}, \mathit{Eps})\}
\end{array}
$$

The diagram produced for our running example is almost the same as before: compared to the one in Fig. 4 we have one additional vertex, labelled $a \cdot (a \mid b)$.



The algorithm has a number of pleasing characteristics. First of all, the number of points and arrows is linear in the size of the given regular expression: the number of intermediate points (states) is bounded by the number of compositions and iterations; the number of arrows (transitions) is bounded by the number of basic symbols plus twice the number of iterations. In practice, the actual size may be significantly smaller through sharing of sub-diagrams. Consider the regular expressions $\mathit{Alt}\ (\mathit{Cat}\ r\ t)\ (\mathit{Cat}\ s\ t)$ and $\mathit{Cat}\ (\mathit{Alt}\ r\ s)\ t$, which are syntactically different but semantically equal by distributivity. Due to the use of sets, the generated diagrams are equal, as well. The sub-diagram for the replicated sub-expression t is automagically shared, as the following calculation demonstrates.

$$
\begin{array}{ll}
 & \mathit{diagram}'\ (\alpha, \mathit{Alt}\ (\mathit{Cat}\ r\ t)\ (\mathit{Cat}\ s\ t), \beta) \\
= & \{\text{ definition of } \mathit{diagram}' \ \} \\
 & \mathit{diagram}'\ (\alpha, r, i) \cup \mathit{diagram}'\ (i, t, \beta) \\
 & \qquad \cup\ \mathit{diagram}'\ (\alpha, s, i) \cup \mathit{diagram}'\ (i, t, \beta)\ \mathbf{where}\ i = \mathit{cat}\ t\ \beta \\
= & \{\text{ idempotency of union } \} \\
 & \mathit{diagram}'\ (\alpha, r, i) \cup \mathit{diagram}'\ (\alpha, s, i) \\
 & \qquad \cup\ \mathit{diagram}'\ (i, t, \beta)\ \mathbf{where}\ i = \mathit{cat}\ t\ \beta \\
= & \{\text{ definition of } \mathit{diagram}' \ \} \\
 & \mathit{diagram}'\ (\alpha, \mathit{Cat}\ (\mathit{Alt}\ r\ s)\ t, \beta)
\end{array}
$$

As another example, $\mathit{Alt}\ r\ r$ and r are mapped to the same diagram. The examples demonstrate that it is undesirable that diagrams for sub-expressions have disjoint states. Sharing of sub-diagrams is a feature, not a bug. Sharing comes at a small cost though: due to the use of sets (rather than lists), the running-time of *diagram* is $\Theta(n \log n)$ rather than linear.

In Sect. 5 we have noted that ϵ-labelled self-loops can be safely removed from a diagram. (Recall that this optimization is justified by $G^* = (G - 1)^*$.)

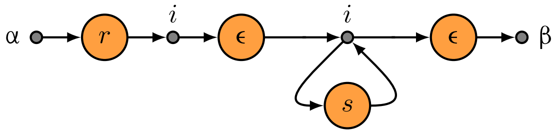
```
trim :: Set (Arrow (Basic a)) → Set (Arrow (Basic a))
trim g = [a | a ← g, not (self-loop a)]

self-loop :: Arrow (Basic a) → Bool
self-loop (i, Eps',    j) = i == j
self-loop (i, Sym' a, j) = False
```

Arrows of the form (r, ϵ, r) are actually not that rare. Our running example features one. More generally, expressions of the form *Cat r* (*Rep s*) generate a self-loop.

$$\begin{aligned}
& \mathit{diagram}'\ (\alpha, \mathit{Cat}\ r\ (\mathit{Rep}\ s), \beta) \\
= & \quad \{\ \text{definition of } \mathit{diagram}'\ \} \\
& \mathit{diagram}'\ (\alpha, r, i) \cup \mathit{diagram}'\ (i, \mathit{Rep}\ s, \beta)\ \mathbf{where}\ i = \mathit{cat}\ (\mathit{Rep}\ s)\ \beta \\
= & \quad \{\ \text{definition of } \mathit{diagram}'\ \} \\
& \mathit{diagram}'\ (\alpha, r, i) \cup \{(i, \mathit{Eps}', i)\} \\
& \qquad \cup\ \mathit{diagram}'\ (i, s, i) \cup \{(i, \mathit{Eps}', \beta)\}\ \mathbf{where}\ i = \mathit{cat}\ (\mathit{Rep}\ s)\ \beta
\end{aligned}$$

The intermediate states for composition and iteration are identified, i, further reducing the total number of states.



7 Finite Automata Without ϵ-Transitions

It is advisable to eliminate ϵ-transitions before translating a nondeterministic finite automaton into an executable program as ϵ-loops might cause termination problems. (Incidentally, Thompson [16] reports that his compilation scheme does not work for $(a^*)^*$ as the code goes into an infinite loop—he also proposes a fix.) There are at least two ways forward.

One option is to apply a graph transformation. Quite pleasingly, the transformation is based on a general property of iteration: $(a + b)^* = (a^* \cdot b)^* \cdot a^*$, see Exercise 5. Let $E = \{(v, \epsilon, w) \mid v, w \in V\}$ be the complete, ϵ-labelled diagram and let G be the diagram generated for r. We can massage the reflexive, transitive closure of G as follows.

$$G^* = ((G \cap E) + (G - E))^* = ((G \cap E)^* \cdot (G - E))^* \cdot (G \cap E)^* \qquad (12)$$

The sub-diagram $C = (G \cap E)^*$ is the so-called ϵ-closure of G. The path-finding problem in G can be reduced to a path-finding problem in $C \cdot (G - E)$:

$$\begin{aligned}
& w \in r \circ\!\!-\!\!(G^*)\!\!-\!\!\circ\, \epsilon \\
\Longleftrightarrow\quad & \{ \text{ see above (12) } \} \\
& w \in r \circ\!\!-\!\!((C \cdot (G - E))^* \cdot C)\!\!-\!\!\circ\, \epsilon \\
\Longleftrightarrow\quad & \{ \text{ composition (11c) } \} \\
& w \in \textstyle\sum \{ (r \circ\!\!-\!\!((C \cdot (G - E))^*)\!\!-\!\!\circ\, f) \cdot (f \circ\!\!-\!\!(C)\!\!-\!\!\circ\, \epsilon) \mid f \in V \} \\
\Longleftrightarrow\quad & \{ \text{ membership } \} \\
& \exists f \in V \,.\, w \in (r \circ\!\!-\!\!((C \cdot (G - E))^*)\!\!-\!\!\circ\, f) \cdot (f \circ\!\!-\!\!(C)\!\!-\!\!\circ\, \epsilon) \\
\Longleftrightarrow\quad & \{ \ q \circ\!\!-\!\!(C)\!\!-\!\!\circ\, z \subseteq \epsilon \text{ and property of free monoid } \} \\
& \exists f \in V \,.\, w \in r \circ\!\!-\!\!((C \cdot (G - E))^*)\!\!-\!\!\circ\, f \quad \wedge \quad \epsilon \in f \circ\!\!-\!\!(C)\!\!-\!\!\circ\, \epsilon
\end{aligned}$$

Let us call a point f with $\epsilon \in f \circ\!\!-\!\!(C)\!\!-\!\!\circ\, \epsilon$ an accepting or final state. The calculation demonstrates that we can reduce the word problem, $w \in [\![r]\!]$, to the problem of finding an w-labelled path from the start state r to an accepting state f in $C \cdot (G - E)$. The final formula explains in a sense why NFAs *without* ϵ-transitions need to have a *set* of final states, whereas a railroad diagram has exactly one entry and one exit.

Another option is to integrate the computation of the ϵ-closure into the construction of the automaton itself. This is exactly what Antimirov's scheme does, which we review next. (The purpose of the following is to show that a railroad diagram contains all of Antimirov's subfactors.)

Antimirov's Linear Forms. To avoid the problems of Brzozowski's construction outlined in Sect. 1, Antimirov devised a special representation of regular expressions based on the notion of a linear form.

$$L - \epsilon = a_1 \cdot L_1 + \cdots + a_n \cdot L_n$$

We represent a linear form by a *set* of pairs, consisting of a symbol and a regular expression.

$$\mathbf{type}\ \mathit{Lin} = \mathit{Set}\ (\mathit{Alphabet}, \mathit{Reg})$$

Observe that the type is non-recursive; the subfactors are still given by regular expressions. Antimirov's insight was that in order to guarantee a finite number of *syntactically* different subfactors, it is sufficient to apply the ACI-properties of choice to the top-level of an expression.

Turning a regular expression into a linear form is straightforward except, perhaps, for composition and iteration. It is *not* the case that the linear form of $r \cdot s - \epsilon$ is given by the composition of the forms for $r - \epsilon$ and $s - \epsilon$. (Do you see why?) The following calculation points us into the right direction.

$$r \cdot s - \epsilon = (r - \epsilon) \cdot s + (r \cap \epsilon) \cdot s - \epsilon = (r - \epsilon) \cdot s + (r \cap \epsilon) \cdot (s - \epsilon)$$

The final formula suggests that we need to compose a linear form, the representation of $r - \epsilon$, with a standard regular expression, namely s.

$$\begin{aligned}
&\textbf{infixr } 7 \bullet \\
&(\bullet) :: \mathit{Lin} \to \mathit{Reg} \to \mathit{Lin} \\
&\mathit{lf} \bullet \mathit{Empty} = \emptyset \\
&\mathit{lf} \bullet \mathit{Eps} \quad\; = \mathit{lf} \\
&\mathit{lf} \bullet s \qquad\;\; = [(a, \mathit{cat}\ r\ s) \mid (a, r) \leftarrow \mathit{lf}]
\end{aligned}$$

Thus, $\mathit{lf} \bullet s$ composes every subfactor of lf with s. (Just in case you wonder, $[e \mid q]$ above is a monad, not a list comprehension.) The calculation for iteration makes use of $a^* = (a - 1)^*$, see Exercise 3.

$$r^* - \epsilon = (r - \epsilon)^* - \epsilon = (r - \epsilon) \cdot (r - \epsilon)^* = (r - \epsilon) \cdot r^*$$

Again, we need to compose a linear form, the representation of $r - \epsilon$, with a standard regular expression, namely r^* itself.

Given these prerequisites, Antimirov's function lf [3], which maps r to the linear form of $r - \epsilon$, can be readily implemented in Haskell.

$$\begin{aligned}
&\mathit{linear\text{-}form} :: \mathit{Reg} \to \mathit{Lin} \\
&\mathit{linear\text{-}form}\ (\mathit{Empty}) = \emptyset \\
&\mathit{linear\text{-}form}\ (\mathit{Alt}\ r\ s) = \mathit{linear\text{-}form}\ r \cup \mathit{linear\text{-}form}\ s \\
&\mathit{linear\text{-}form}\ (\mathit{Eps}) \quad\;\; = \emptyset \\
&\mathit{linear\text{-}form}\ (\mathit{Sym}\ a) \;\; = \{(a, \mathit{Eps})\} \\
&\mathit{linear\text{-}form}\ (\mathit{Cat}\ r\ s) \\
&\quad \mid \mathit{nullable}\ r \qquad = \mathit{linear\text{-}form}\ r \bullet s \cup \mathit{linear\text{-}form}\ s \quad \text{-- } r \cap \epsilon = \epsilon \\
&\quad \mid \mathit{otherwise} \qquad = \mathit{linear\text{-}form}\ r \bullet s \qquad\qquad\qquad\quad\;\; \text{-- } r \cap \epsilon = \emptyset \\
&\mathit{linear\text{-}form}\ (\mathit{Rep}\ r) \;\; = \mathit{linear\text{-}form}\ r \bullet \mathit{Rep}\ r
\end{aligned}$$

For our running example, we obtain

$$\begin{aligned}
&\ggg \quad \mathit{linear\text{-}form}\ (\mathit{Cat}\ (\mathit{Rep}\ ab)\ (\mathit{Cat}\ a\ ab)) \\
&\{(\texttt{'a'}, ab), (\texttt{'a'}, \mathit{Cat}\ (\mathit{Rep}\ ab)\ (\mathit{Cat}\ a\ ab)), (\texttt{'b'}, \mathit{Cat}\ (\mathit{Rep}\ ab)\ (\mathit{Cat}\ a\ ab))\}
\end{aligned}$$

Antimirov's Subfactors. Putting the automata glasses on, *linear-form* r computes the outgoing edges of r. The successor states of r are given by the immediate subfactors.

$$\mathit{subfactors}\ r = [\beta \mid (a, \beta) \leftarrow \mathit{linear\text{-}form}\ r] \qquad (13)$$

In order to easily compare Antimirov's construction to Manna's, it is useful to concentrate on subfactors. To this end, we unfold the specification (13) to obtain

```
$subfactors :: Reg \rightarrow Set\ Reg$
$subfactors\ (Empty) = \emptyset$
$subfactors\ (Alt\ r\ s) = subfactors\ r \cup subfactors\ s$
$subfactors\ (Eps) \quad = \emptyset$
$subfactors\ (Sym\ a) = \{Eps\}$
$subfactors\ (Cat\ r\ s)$
    $\mid nullable\ r \qquad = subfactors\ r \circ s \cup subfactors\ s$
    $\mid otherwise \qquad = subfactors\ r \circ s$
$subfactors\ (Rep\ r) \quad = subfactors\ r \circ Rep\ r$
```

The operator "∘" is a version of "•" that works on sets of states, rather than sets of edges.

```
**infixr** 7 $\circ$
$(\circ) :: Set\ Reg \rightarrow Reg \rightarrow Set\ Reg$
$rs \circ Empty = \emptyset$
$rs \circ Eps \quad = rs$
$rs \circ s \qquad = [cat\ r\ s \mid r \leftarrow rs]$
```

The reflexive, transitive closure of *subfactors* applied to the given regular expression r then yields the states of the Antimirov automaton: starting with $\{r\}$ we iterate $subfactors^{\dagger}$ until a fixed-point is reached, where $f^{\dagger}$ is the so-called Kleisli extension of f, defined $f^{\dagger}\ X = \bigcup\{\, f\ x \mid x \in X\,\}$. That is easy enough—however, it is, perhaps, not immediately clear that the set of all subfactors, immediate and transitive ones, is finite.

Manna's Construction Revisited. We claim that Antimirov's subfactors appear as target states of non-ϵ-transitions in the corresponding railroad diagram. Given the specification,

$$targets\ r = [\beta \mid (\alpha, Sym'\ a, \beta) \leftarrow diagram\ r]$$

it is a straightforward exercise to derive $targets\ r = targets'\ (r, Eps)$ where $targets'$ is defined

```
$targets' :: (Reg, Reg) \rightarrow Set\ Reg$
$targets'\ (Empty,\ \beta) = \emptyset$
$targets'\ (Alt\ r\ s,\ \beta) = targets'\ (r, \beta) \cup targets'\ (s, \beta)$
$targets'\ (Eps, \beta) \qquad = \emptyset$
$targets'\ (Sym\ a,\ \beta) = \{\beta\}$
$targets'\ (Cat\ r\ s, \beta) = targets'\ (r, cat\ s\ \beta) \cup targets'\ (s, \beta)$
$targets'\ (Rep\ r,\ \beta) = targets'\ (r, cat\ (Rep\ r)\ \beta)$
```

This is basically the definition of *diagram'*, only that we ignore source points and labels and discard ϵ-labelled arrows.

The definition of *targets'* is tantalizingly close to *subfactors*, except that the former makes use of an accumulating parameter, whereas the latter does not. Removing the accumulating parameter is, of course, a matter of routine.

$$\begin{aligned}
&\mathit{targets} :: \mathit{Reg} \to \mathit{Set}\ \mathit{Reg} \\
&\mathit{targets}\ (\mathit{Empty}) &&= \emptyset \\
&\mathit{targets}\ (\mathit{Alt}\ r\ s) &&= \mathit{targets}\ r \cup \mathit{targets}\ s \\
&\mathit{targets}\ (\mathit{Eps}) &&= \emptyset \\
&\mathit{targets}\ (\mathit{Sym}\ a) &&= \{\mathit{Eps}\} \\
&\mathit{targets}\ (\mathit{Cat}\ r\ s) &&= \mathit{targets}\ r \circ s \cup \mathit{targets}\ s \\
&\mathit{targets}\ (\mathit{Rep}\ r) &&= \mathit{targets}\ r \circ \mathit{Rep}\ r
\end{aligned}$$

(The transformation is only meaning-preserving if the accumulation is based on a monoid. Alas, concatenation of regular expressions is not associative, so *targets* and *targets′* produce sets of syntactically different regular expressions that are, however, semantically equivalent: $[\![\mathit{targets}'\ (r, \beta)]\!] = [\![\mathit{targets}\ r \circ \beta]\!]$. We choose to ignore this technicality.)

Theorem 2. *Antimirov's subfactors are contained in Manna's automaton.*

$$\mathit{subfactors}\ r \subseteq \mathit{targets}\ r \tag{14a}$$

$$\mathit{targets}^{\dagger}\ (\mathit{targets}\ r) \subseteq \mathit{targets}\ r \tag{14b}$$

Recall that $f^{\dagger}$ is the Kleisli extension of f to sets.

We first establish the following properties of *targets*.

$$\mathit{targets}\ (\mathit{cat}\ r\ s) \subseteq \mathit{targets}\ r \circ s \cup \mathit{targets}\ s \tag{14c}$$

$$\mathit{targets}^{\dagger}\ (\mathit{rs}\ \circ\ s) \subseteq \mathit{targets}^{\dagger}\ \mathit{rs} \circ s \cup \mathit{targets}\ s \tag{14d}$$

The proof of (14c) is straightforward and omitted. (If r is non-null, the inequality can be strengthened to an equality.) For (14d) we reason,

$$\begin{aligned}
&\quad \mathit{targets}^{\dagger}\ (\mathit{rs} \circ s) \\
&= \quad \{ \text{ definition of } f^{\dagger} \ \} \\
&\quad \bigcup \{ \mathit{targets}\ (\mathit{cat}\ r\ s) \mid r \in \mathit{rs} \} \\
&\subseteq \quad \{ \text{ property of of } \mathit{targets} \text{ (14c) } \} \\
&\quad \bigcup \{ \mathit{targets}\ r \circ s \cup \mathit{targets}\ s \mid r \in \mathit{rs} \} \\
&= \quad \{ \text{ set union } \} \\
&\quad \bigcup \{ \mathit{targets}\ r \circ s \mid r \in \mathit{rs} \} \cup \mathit{targets}\ s \\
&= \quad \{ \text{ “}\circ\text{” distributes over set union } \} \\
&\quad \bigcup \{ \mathit{targets}\ r \mid r \in \mathit{rs} \} \circ s \cup \mathit{targets}\ s \\
&= \quad \{ \text{ definition of } f^{\dagger} \ \} \\
&\quad \mathit{targets}^{\dagger}\ \mathit{rs} \circ s \cup \mathit{targets}\ s
\end{aligned}$$

Proof (Theorem 2). It is fairly obvious that *subfactors r* is a subset of *targets r* as only the clause for concatenation is different. The proof of (14b) proceeds by induction of the structure of r.

Case *Empty* **and** *Eps***:**

$$\begin{aligned}
& \mathit{targets}^{\dagger}\ (\mathit{targets}\ \mathit{Empty}) \\
=\ & \quad \{\ \text{definition of } \mathit{targets}\ \} \\
& \mathit{targets}^{\dagger}\ \emptyset \\
=\ & \quad \{\ \text{definition of } f^{\dagger}\ \} \\
& \emptyset
\end{aligned}$$

Case *Alt r s***:**

$$\begin{aligned}
& \mathit{targets}^{\dagger}\ (\mathit{targets}\ (\mathit{Alt}\ r\ s)) \\
=\ & \quad \{\ \text{definition of } \mathit{targets}\ \} \\
& \mathit{targets}^{\dagger}\ (\mathit{targets}\ r \cup \mathit{targets}\ s) \\
=\ & \quad \{\ f^{\dagger} \text{ distributes over set union}\ \} \\
& \mathit{targets}^{\dagger}\ (\mathit{targets}\ r) \cup \mathit{targets}^{\dagger}\ (\mathit{targets}\ s) \\
\subseteq\ & \quad \{\ \text{induction assumption and monotonicity}\ \} \\
& \mathit{targets}\ r \cup \mathit{targets}\ s \\
=\ & \quad \{\ \text{definition of } \mathit{targets}\ \} \\
& \mathit{targets}\ (\mathit{Alt}\ r\ s)
\end{aligned}$$

Case *Sym a***:**

$$\begin{aligned}
& \mathit{targets}^{\dagger}\ (\mathit{targets}\ (\mathit{Sym}\ a)) \\
=\ & \quad \{\ \text{definition of } \mathit{targets}\ \} \\
& \mathit{targets}^{\dagger}\ \{\mathit{Eps}\} \\
=\ & \quad \{\ \text{definition of } f^{\dagger}\ \} \\
& \bigcup\{\mathit{targets}\ \mathit{Eps}\} \\
=\ & \quad \{\ \text{definition } \mathit{targets} \text{ and } \bigcup\{\emptyset\} = \emptyset\ \} \\
& \emptyset
\end{aligned}$$

Case *Cat r s*:

$$
\begin{array}{cl}
& \mathit{targets}^{\dagger}\ (\mathit{targets}\ (\mathit{Cat}\ r\ s)) \\
= & \{\ \text{definition of } \mathit{targets}\ \} \\
& \mathit{targets}^{\dagger}\ (\mathit{targets}\ r \circ s \cup \mathit{targets}\ s) \\
= & \{\ f^{\dagger} \text{ distributes over set union }\} \\
& \mathit{targets}^{\dagger}\ (\mathit{targets}\ r \circ s) \cup \mathit{targets}^{\dagger}\ (\mathit{targets}\ s) \\
= & \{\ (14\text{d})\ \} \\
& \mathit{targets}^{\dagger}\ (\mathit{targets}\ r) \circ s \cup \mathit{targets}\ s \cup \mathit{targets}^{\dagger}\ (\mathit{targets}\ s) \\
\subseteq & \{\ \text{induction assumption and monotonicity }\} \\
& \mathit{targets}\ r \circ s \cup \mathit{targets}\ s \cup \mathit{targets}\ s \\
= & \{\ \text{definition of } \mathit{targets} \text{ and idempotency }\} \\
& \mathit{targets}\ (\mathit{Cat}\ r\ s)
\end{array}
$$

Case *Rep r*:

$$
\begin{array}{cl}
& \mathit{targets}^{\dagger}\ (\mathit{targets}\ (\mathit{Rep}\ r)) \\
= & \{\ \text{definition of } \mathit{targets}\ \} \\
& \mathit{targets}^{\dagger}\ (\mathit{targets}\ r \circ \mathit{Rep}\ r) \\
= & \{\ (14\text{d})\ \} \\
& \mathit{targets}^{\dagger}\ (\mathit{targets}\ r) \circ \mathit{Rep}\ r \cup \mathit{targets}\ (\mathit{Rep}\ r) \\
\subseteq & \{\ \text{induction assumption and monotonicity }\} \\
& \mathit{targets}\ r \circ \mathit{Rep}\ r \cup \mathit{targets}\ (\mathit{Rep}\ r) \\
= & \{\ \text{definition of } \mathit{targets} \text{ and idempotency }\} \\
& \mathit{targets}\ (\mathit{Rep}\ r)
\end{array}
$$

We may conclude that a regular expression r has only a finite number of *syntactically* different right subfactors, as each subfactor appears as a target state in the corresponding railroad diagram. Moreover, subfactors have a very simple structure: they are compositions of sub-expressions of r.

Just in case you wonder, the converse is not true—not every target is also an Antimirov subfactor. If the regular expression contains *Empty* as a sub-expression, then Manna's automaton may contain unreachable states. Consider, for example, *Cat* (*Cat Empty a*) *b*.

8 Related Work

Manna's Generalized Transition Graphs. Our diagrams are modelled after Manna's generalized transition graphs, directed graphs labelled with regular expressions. For the discussion, it is useful to remind us of the different ways of defining labelled graphs.

$$\begin{array}{ccccccc}
\text{labelled graph} & & \text{state-transition} & & \text{adjacency list} & & \text{adjacency matrix} \\
\mathcal{P}(V \times \Sigma \times V) & \cong & V \times \Sigma \to \mathcal{P}(V) & \cong & V \to \mathcal{P}(\Sigma \times V) & \cong & V \times V \to \mathcal{P}(\Sigma) \\
(i, a, j) \in G & & \delta(i, a) \ni j & & \mathit{lf}\ i \ni (a, j) & & i \mathrel{\circ\!\!-\!\!\!-\!\!\circ} j \ni a
\end{array}$$

The isomorphisms are based on the one-to-one correspondence between relations and set-valued functions, $\mathcal{P}(A \times B) \cong A \to \mathcal{P}(B)$. Each "view" is in use. Manna models deterministic finite automata as labelled graphs with certain restrictions on the edges to ensure determinacy. The standard definition of nondeterministic finite automata is based on state-transition functions [15]. The adjacency list representation underlies Antimirov's linear forms where $V := \mathcal{P}(\Sigma)$, see Sect. 7. The adjacency matrix representation emphasizes the generative nature of automata, see Sect. 5. (Actually, there is also a fifth alternative, $\Sigma \to \mathcal{P}(V \times V)$, which, however, does not seem to be popular.)

The design space has a further dimension: we can equip the type of labels with structure. Manna first generalizes DFAs to transition graphs by allowing words as labels and then to generalized transition graphs, where arrows are labelled with regular expressions.

Σ	symbol	NFA
$\{\epsilon\} \cup \Sigma$	basic symbol	NFA with ϵ-transitions
Σ^*	word	Manna's transition graphs
$\mathcal{P}(\Sigma^*)$	language	Manna's generalized transition graphs

Generalizing Σ^* to an arbitrary monoid, we obtain the diagrams of Sect. 5, which form a regular algebra. They are isomorphic to Backhouse's matrices [4], where the underlying regular algebra is given by a lifted monoid.

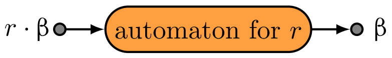
$$\begin{array}{ccc}
\text{diagram} & & \text{matrix} \\
\mathcal{P}(V \times M \times V) & \cong & V \times V \to \mathcal{P}(M) \\
(i, w, j) \in G & & i \mathrel{\circ\!\!-\!\!\!-\!\!\circ} j \ni w
\end{array}$$

The isomorphism is also a regular homomorphism.

The McNaughton-Yamada-Thompson Algorithm. One of the first algorithms for converting a regular expression to a DFA is due to McNaughton and Yamada [13]. Though not spelled out explicitly, their algorithm first constructs an NFA without ϵ-transitions, which is subsequently converted to a DFA using the subset construction. In the first phase, they annotate each symbol of the regular expression with a position, which corresponds roughly to a station and its target point in our setting. The NFA is obtained by suitably connecting "terminal" and "initial" positions of sub-expressions for composition and iteration. Nonetheless, the idea of using an NFA as an intermediary is usually attributed to Thompson [16].

A standard textbook algorithm, the McNaughton-Yamada-Thompson algorithm [2], is based on their ideas. The algorithm proceeds by induction over the structure of the regular expression, as illustrated in Fig. 5. Each automaton has one start state (with no outgoing transitions) and one accepting state (with no incoming transitions). To avoid interference, states must be suitably renamed when sub-automata are combined. (The exact nature of the states is actually somewhat unclear—in the illustration they are not even named.) The resulting automata feature quite a few ϵ-transitions, not all of which are present in the original articles.

It is instructive to scrutinize Thompson's algorithm for "regular expression search" [16]. He explains the workings of his compiler using $a \cdot (b \mid c) \cdot d$ as a running example, illustrating the steps with diagrams. Interestingly, the illustrations are quite close to railway diagrams. Consequently, his algorithm can be easily recast in our framework, see Fig. 6. Thompson uses ϵ-transitions only for choice (and iteration which involves choice). Renaming of states is necessary for sub-automata that do not end in ϵ.



In this case, β must be appended to each state. Turning to an implementation in Haskell, the renaming operation has a familiar ring.

$$
\begin{array}{lll}
\textbf{infixr } 7 \odot & & \\
(\odot) :: \mathit{Set}\ (\mathit{Arrow}\ a) \to \mathit{Reg} \to \mathit{Set}\ (\mathit{Arrow}\ a) & & \\
g \odot \mathit{Empty} & = \emptyset & \\
g \odot \mathit{Eps} & = g & \\
g \odot s & = [(\mathit{cat}\ \alpha\ s, r, \mathit{cat}\ \beta\ s) \mid (\alpha, r, \beta) \leftarrow g] &
\end{array}
$$

We have introduced similar operations for Antimirov's linear forms and subfactors. Observe that the diagram $G \odot \beta$ is admissible if G is.

Thompson's translation is then captured by the following Haskell program (ignoring the fact that his compiler actually produces IBM 7094 machine code).

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{thompson} :: \mathit{Reg} \to \mathit{Set}\ (\mathit{Reg}, \mathit{Basic}\ \mathit{Alphabet}, \mathit{Reg})} \\
\mathit{thompson}\ \mathit{Eps} & = \{(\mathit{Eps}, \mathit{Eps}', \mathit{Eps})\} \\
\mathit{thompson}\ (\mathit{Sym}\ a) & = \{(\mathit{Sym}\ a, \mathit{Sym}'\ a, \mathit{Eps})\} \\
\mathit{thompson}\ (\mathit{Cat}\ r\ s) & = \mathit{thompson}\ r \odot s \cup \mathit{thompson}\ s \\
\mathit{thompson}\ \mathit{Empty} & = \emptyset \\
\mathit{thompson}\ (\mathit{Alt}\ r\ s) & = \{(\mathit{Alt}\ r\ s, \mathit{Eps}', r)\} \\
 & \cup\ \{(\mathit{Alt}\ r\ s, \mathit{Eps}', s)\} \cup \mathit{thompson}\ r \cup \mathit{thompson}\ s \\
\mathit{thompson}\ (\mathit{Rep}\ r) & = \{(\mathit{Rep}\ r, \mathit{Eps}', \mathit{Eps})\} \\
 & \cup\ \{(\mathit{Rep}\ r, \mathit{Eps}', \mathit{cat}\ r\ (\mathit{Rep}\ r))\} \cup \mathit{thompson}\ r \odot \mathit{Rep}\ r
\end{array}
$$

Even though similar in appearance, Thompson's compiler is quite different from Manna's construction. Manna's algorithm is *iterative* or top-down: *diagram* iteratively replaces a single arrow by a smallish diagram. Thompson's algorithm is *recursive* or bottom-up: *thompson* recursively combines sub-automata for subexpressions. For composition (and iteration which involves composition), this requires explicit renaming of states. In Manna's construction renaming is, in some sense, implicit through the use of an accumulating parameter. (As an aside, *thompson* also enables sharing of sub-automata as renaming does not operate on anonymous states, but on subfactors, which are semantically meaningful; consider, for example, $\mathit{Alt}\ (\mathit{Cat}\ r\ t)\ (\mathit{Cat}\ s\ t)$ and $\mathit{Cat}\ (\mathit{Alt}\ r\ s)\ t$.)

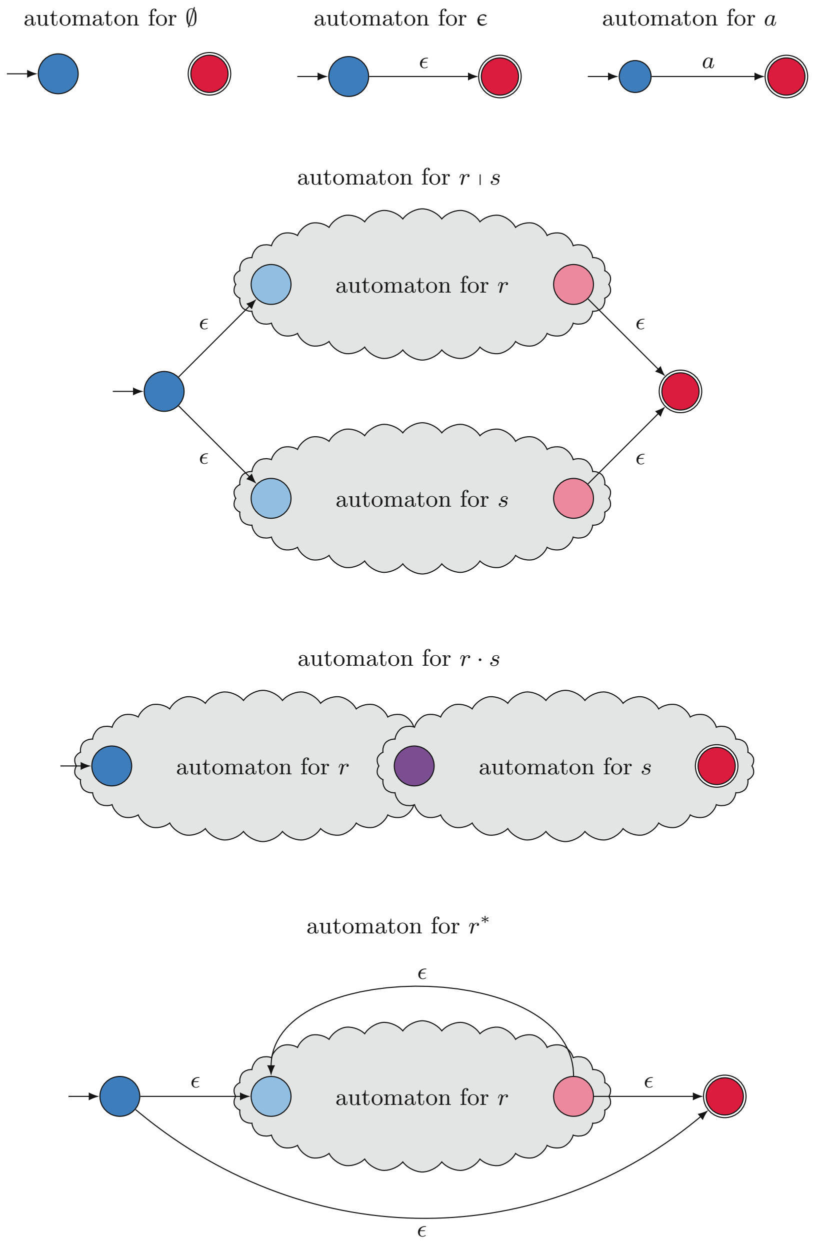


Fig. 5. McNaughton-Yamada-Thompson construction of an NFA with ϵ-transitions.

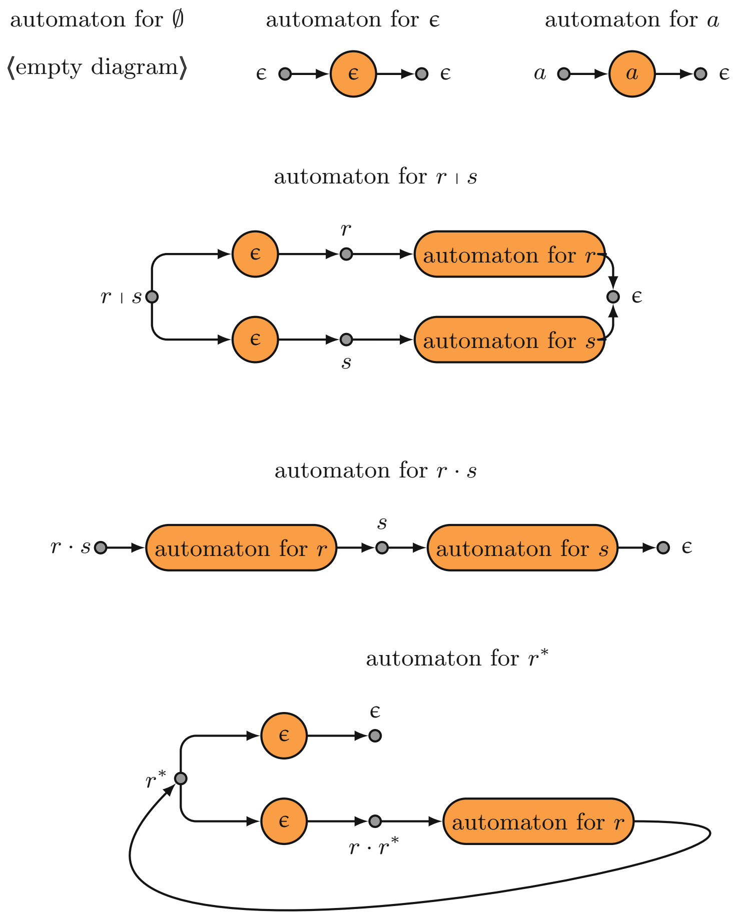


Fig. 6. Thompson's "original" construction of an NFA with ϵ-transitions.

Conways's Linear Mechanisms. A related construction is given by Conway [8]. In his seminal book on "Regular Algebra and Finite Machines" he defines a *linear mechanism*, which amounts to a nondeterministic finite automaton with ϵ-transitions. Interestingly, his automata feature both multiple final states *and* multiple start states, which allows for a very symmetric treatment.

He represents an automaton by an $n \times n$ matrix, where n is the number of vertices. To illustrate, the automaton of Fig. 3 is captured by

$$S = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \qquad M = \begin{pmatrix} 0 & 1 & 0 & 0 \\ a+b & 0 & a & 0 \\ 0 & 0 & 0 & a+b \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad F = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

The Boolean row vector S determines the start states; the Boolean column vector F determines the final states; the transitions are represented by the square matrix M. The language generated by the automaton is then given by SM^*F, where M^* is the reflexive, transitive closure of M. On a historical note, Conway [8] mentions that the matrix formula was already known to P. J. Cleave in 1961. The fact that matrices form a regular algebra was probably apparent to Conway, even though he did not spell out the details. Nonetheless, Conway's work had a major influence on Backhouse's treatment of regular algebra (personal communication).

Like Thompson's algorithm, the translation of regular expressions proceeds *recursively* or *inductively*.

$$\begin{aligned}
0 &= 0\begin{pmatrix}0\end{pmatrix}^* 0\\
1 &= 1\begin{pmatrix}1\end{pmatrix}^* 1\\
a &= \begin{pmatrix}1 & 0\end{pmatrix}\begin{pmatrix}0 & a\\ 0 & 0\end{pmatrix}^*\begin{pmatrix}0\\ 1\end{pmatrix}\\
SM^*F + TN^*G &= \begin{pmatrix}S & T\end{pmatrix}\begin{pmatrix}M & 0\\ 0 & N\end{pmatrix}^*\begin{pmatrix}F\\ G\end{pmatrix}\\
SM^*F \cdot TN^*G &= \begin{pmatrix}S & 0\end{pmatrix}\begin{pmatrix}M & FT\\ 0 & N\end{pmatrix}^*\begin{pmatrix}0\\ G\end{pmatrix}\\
(SM^*F)^* &= \begin{pmatrix}0 & 1\end{pmatrix}\begin{pmatrix}M & F\\ S & 0\end{pmatrix}^*\begin{pmatrix}0\\ 1\end{pmatrix}
\end{aligned}$$

The automata for the empty language and the empty word feature a single state; the automaton for a symbol has two states. For composition, choice, and iteration we assume that the sub-expressions are already translated into automata. The combined automata are then expressed by suitable block matrices that detail the interaction of the component automata. For example, the automaton for iteration r^* adds one state to the automaton for r; this state is both a start and a final state; there is an ϵ-transition from the new state to each start state of r and from each final state of r to the new state.

Like the McNaughton-Yamada-Thompson construction, Conway's automata feature quite a few ϵ-transitions as there is no sharing of sub-automata. For example, the automaton for composition adds an ϵ-transition from each final state of the first to each initial state of the second automaton (represented by the matrix FT). On the positive side, the correctness of the translation can be readily established using the following characterization of iteration [8].

$$\begin{pmatrix}A & B\\ C & D\end{pmatrix}^* = \begin{pmatrix}X^* & A^*BY^*\\ D^*CX^* & Y^*\end{pmatrix} \quad \textbf{where} \quad \begin{cases}X = A + BD^*C\\ Y = D + CA^*B\end{cases}$$

The block matrix consists of two square matrices A and D, which represent sub-automata, and two rectangular matrices B and C, which record transitions between the sub-automata. The entries on the right specify the possible paths in the combined automaton, for example, $A^* B Y^*$ contains all paths from A to D: a path in A (ie A^*), followed by an edge from A to D (ie B), followed by a path from D to D (ie Y^*). Given this decomposition, the correctness of the translation can be shown using straightforward equational reasoning.

Brzozowski's Factors and Antimirov's Subfactors. Brzozowski [6] showed that a regular expression has only a finite number of *syntactically* different factors, if expressions are compared modulo associativity, commutativity, and idempotence of choice. Antimirov [3] pointed out that some care has to be exercised when computing the factors. Brzozowski uses the following formula for composition: $a \setminus (r \cdot s) = (a \setminus r) \cdot s + \delta\, r \cdot (a \setminus s)$, where $\delta\, r = 1$ if r is nullable and $\delta\, r = 0$ otherwise. To ensure finiteness, the definition of δ must actually be unfolded: $a \setminus (r \cdot s) = \textbf{if}\ \mathit{nullable}\ r\ \textbf{then}\ (a \setminus r) \cdot s + (a \setminus s)\ \textbf{else}\ (a \setminus r) \cdot s$.

Antimirov further realized that it is sufficient to apply the ACI-properties of choice only to the top-level of a term, see his definition of linear form. His approach essentially derives a system of equations of the form (2) from a regular expression. Based on the same idea, Mirkin [14] gave an algorithm for constructing an NFA, predating Antimirov's work by almost two decades. Champarnaud and Ziadi [7] pointed out the similarity, attempting to show that the two approaches actually yield the same automata. Unfortunately, their proof contains an error, which was later corrected by Broda et al. [5]. In more detail, Champarnaud and Ziadi claim that the function *subfactors*, which they call π, computes the immediate *and* transitive subfactors, whereas it only determines the former. Broda et al. pointed out that *subfactors* must be replaced by *targets*. (Contrary to their claim, even then the two constructions are not identical as pointed out in Sect. 7: *targets* may include unreachable states not present in Antimirov's construction. A minor technicality, which can be fixed by excluding *Empty* as a constructor.) The corrected definition of π seems to fall out of thin air though—it is pleasing to see that it is obtained as a projection of Manna's automaton.

9 Conclusion

Regular algebra is the algebra of three operators central to programming: composition, choice, and iteration. As such, it is perhaps the most fundamental algebraic structure in computing science.

Roland Backhouse

I could not agree more. We have used regular algebra to reason both about languages and diagrams. Students of computing science should see at least a glimpse of regular algebra in their first term. Regular expressions and railroad diagrams provide an ideal starting point. Manna's construction, which ties the two concepts together, is both charming and challenging. It is charming because the transformations are local, supporting an iterative, step by step refinement of diagrams. It is challenging for the same reason: one has to ensure that different parts of the diagram do not interfere. This is where subfactors (called partial derivatives elsewhere) enter the scene. Standard textbook proofs of the equivalence of regular expressions and nondeterministic finite automata often involve verbose arguments, making implicit assumptions about disjointness of state sets ("Here, i is a new state, ..."). By contrast, subfactors facilitate simple, calculational correctness proofs, based on fundamental properties of Galois connections. What equational reasoning with factors is for DFAs, inequational reasoning with subfactors is for NFAs.

(On a personal note, I think that it is a mistake to introduce a finite automaton in this particular context as a quintuple $(\Sigma, S, s_0, F, \delta)$ where S is some anonymous, unstructured set of states. Subfactors as states serve as important scaffolding that should only be removed in a final abstraction step—once Kleene's Theorem is established or other uses for finite automata have been introduced.)

Despite its simplicity, Manna's construction has a number of pleasing characteristics: the number of states and the number of edges is linear in the size of the regular expression; due to sharing of sub-automata and auto-merging of states the resulting automaton is often surprisingly small. This demonstrates that disjointness of state sets is undesirable or, put differently, "renaming" should be semantically meaningful: "$\bullet$" and "$\odot$" can be seen as incarnations of the distributive law.

Finally, it has been satisfying to be able to relate Manna's construction to Antimirov's subfactors through simple program transformations, based on accumulating parameters.

Acknowledgements. A big thank you is due to Bernhard Möller for pointing me to Manna's textbook on the "Mathematical Theory of Computation" [12]. Thanks are also due to Clare Martin and Sebastian Schweizer for proof-reading a preliminary version of the manuscript. The Ti*k*Z diagrams for Thompson's construction, see Fig. 5, are due to Ian McLoughlin, who kindly granted permission to use them. As always, the anonymous referees of MPC provided detailed feedback, suggesting numerous improvements regarding presentation and organization of the material. In response to their comments I moved the section on "Railroad Diagrams" to the front and I clarified Antimirov's contribution. Finally, I owe a particular debt of gratitude to Roland Backhouse, who provided numerous comments, pointing out, in particular, the relevance of Conway's work. The paragraph on "linear mechanisms" was added in response to his suggestions.

References

1. Abramsky, S., Vickers, S.: Quantales, observational logic and process semantics. Math. Struct. Comput. Sci. **3**(2), 161–227 (1993). https://doi.org/10.1017/S0960129500000189
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools, 2nd edn. Pearson Addison-Wesley, Boston (2007)

3. Antimirov, V.: Partial derivatives of regular expressions and finite automaton constructions. Theor. Comput. Sci. **155**(2), 291–319 (1996). https://doi.org/10.1016/0304-3975(95)00182-4. http://www.sciencedirect.com/science/article/pii/0304397595001824
4. Backhouse, R.: Regular algebra applied to language problems. J. Logic Algebraic Program. **66**(2), 71–111 (2006). https://doi.org/10.1016/j.jlap.2005.04.008. http://www.sciencedirect.com/science/article/pii/S1567832605000329
5. Broda, S., Machiavelo, A., Moreira, N., Reis, R.: On the average state complexity of partial derivative automata: an analytic combinatorics approach. Int. J. Found. Comput. Sci. **22**(07), 1593–1606 (2011). https://doi.org/10.1142/S0129054111008908
6. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4), 481–494 (1964). https://doi.org/10.1145/321239.321249. http://doi.acm.org/10.1145/321239.321249
7. Champarnaud, J.M., Ziadi, D.: From Mirkin's prebases to Antimirov's word partial derivatives. Fundam. Inf. **45**(3), 195–205 (2001)
8. Conway, J.H.: Regular Algebra and Finite Machines. Chapman and Hall, London (1971)
9. Jensen, K., Wirth, N.: Pascal: User Manual and Report, 2nd edn. Springer, Heidelberg (1978)
10. Kleene, S.C.: Representation of events in nerve nets and finite automata. Technical report, RM-704, U.S. Air Force, Project RAND, Research Memorandum, December 1951
11. Mac Lane, S.: Categories for the Working Mathematician. Graduate Texts in Mathematics, 2nd edn. Springer, Heidelberg (1998)
12. Manna, Z.: Introduction to Mathematical Theory of Computation. McGraw-Hill Book Company, New York (1974)
13. McNaughton, R., Yamada, H.: Regular expressions and state graphs for automata. IRE Trans. Electron. Comput. **EC-9**(1), 39–47 (1960). https://doi.org/10.1109/TEC.1960.5221603
14. Mirkin, B.G.: An algorithm for constructing a base in a language of regular expressions. Eng. Cybern. **5**, 110–116 (1966)
15. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM J. Res. Dev. **3**(2), 114–125 (1959). https://doi.org/10.1147/rd.32.0114
16. Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM **11**(6), 419–422 (1968). https://doi.org/10.1145/363347.363387. http://doi.acm.org/10.1145/363347.363387