Unified Decision Procedures for Regular Expression Equivalence

Tobias Nipkow and Dmitriy Traytel

Fakultät für Informatik, Technische Universität München, Germany

Abstract. We formalize a unified framework for verified decision procedures for regular expression equivalence. Five recently published formalizations of such decision procedures (three based on derivatives, two on marked regular expressions) can be obtained as instances of the framework. We discover that the two approaches based on marked regular expressions, which were previously thought to be the same, are different, and we prove a quotient relation between the automata produced by them. The common framework makes it possible to compare the performance of the different decision procedures in a meaningful way.

1 Introduction

Equivalence of regular expressions is a perennial topic in computer science. Recently it has spawned a number of formalized and verified decision procedures for this task in interactive theorem provers [3, 6, 10, 19, 21]. Except for the formalization by Braibant and Pous [6], all these decision procedures operate directly on variations of regular expressions. Although they (implicitly) build automata, the states of the automata are labeled with regular expressions, and there is no global transition table but the next-state function is computable from the regular expressions. The motivation for working with regular expressions is simplicity: regular expressions are a free datatype which proof assistants and their users love because it means induction, recursion and equational reasoning—the core competence of proof assistants and functional programming languages. Yet all these decision procedures based on regular expressions look very different. Of course, the next-state functions all differ, but so do the actual decision procedures and their correctness, completeness and termination proofs. The contributions of our paper are the following:

- A unified framework (Sect. 3) that we instantiate with all the above approaches (Sects. 4 and 5). The framework is a simple reflexive transitive closure computation that enumerates the states of a product automaton.
- Proofs of correctness, completeness and termination that are performed once and for all for the framework based on a few properties of the next-state function.
- A new perspective on partial derivatives that recasts them as Brzozowski derivatives followed by some rewriting (Sect. 4).
- The discovery that Asperti's algorithm is not the one by McNaughton-Yamada [20], as stated by Asperti [3], but a dual construction which apparently had not been considered in the literature and which produces smaller automata (Sect. 5).
- An empirical comparison of the performance of the different approaches (Sect. 6).

The discussion of related work is distributed over the relevant sections of the paper.

2 Preliminaries

Isabelle/HOL is based on Church's simple type theory (see [22, Part I] for a recent introduction). Types τ are built from type variables α, β, etc. via function types, other type constructors are written postfix. The notation $t :: \tau$ means that term t has type τ. Types $\alpha\,set$ and $\alpha\,list$ are the types of sets and lists of elements of type α. They come with the following vocabulary: function set (conversion from lists to sets), $[]$ (empty list), # (list constructor), @ (append), hd (head), tl (tail) and map.

Recursive functions over datatypes are executable, and Isabelle can generate from them code in functional languages [15]. This includes functions on finite sets [14]. Unless stated otherwise all functions in this paper are executable.

Locales [4] are Isabelle's tool for modelling parameterized systems. A locale fixes parameters and states assumptions about them:

$$\textbf{locale}\ A = \textbf{fixes}\ x_1\ \textbf{and}\ \ldots\ \textbf{and}\ x_n\ \textbf{assumes}\ n_1 : P_1\ \overline{x}\ \textbf{and}\ \ldots\ \textbf{and}\ n_m : P_m\ \overline{x}$$

In the context of the locale A, we can define constants that depend on the parameters x_i and prove properties about those constants using the assumptions P_i (accessed under the name n_i). Parameters can be instantiated: **interpretation** $\mathsf{J}: A$ **where** $x_1 = t_1 \ldots x_n = t_n$. The command issues proof obligations $P_i\ \bar{t}$ (that the user must discharge) and exports constants and theorems from the locale with x_i instantiated to t_i. Multiple interpretations of the same locale are possible; the prefix "J." disambiguates different instances.

Regular expressions are defined as a recursive datatype:

$$\textbf{datatype}\ \alpha\,rexp = \mathbf{0} \mid \mathbf{1} \mid \mathsf{A}\,\alpha \mid \alpha\,rexp + \alpha\,rexp \mid \alpha\,rexp \cdot \alpha\,rexp \mid (\alpha\,rexp)^*$$

with the usual (non-executable) semantics $\mathscr{L} :: \alpha\,rexp \rightarrow \alpha\,lang$, where $\alpha\,lang$ is short for $(\alpha\,list)\,set$. In concrete regular expressions, we sometimes omit the constructor A for readability. The recursive function $\mathsf{nullable} :: \alpha\,rexp \rightarrow bool$ satisfies $\mathsf{nullable}\ r \longleftrightarrow [] \in \mathscr{L} r$. The functions $\Sigma :: \alpha\,rexp \rightarrow \alpha\,set$ and $\mathsf{atoms} :: \alpha\,rexp \rightarrow \alpha\,list$ compute the set and list of atoms (the arguments of constructor A) in a regular expression. The (non-executable) *left quotient* of a language $L :: \alpha\,lang$ w.r.t. some $a :: \alpha$ is defined by $\mathscr{D}\ a\ L = \{w \mid a \# w \in L\}$. The extension of $\mathscr{D}$ from single symbols to words $w :: \alpha\,list$ can be expressed as $\mathsf{fold}\ \mathscr{D}\ w$ where $\mathsf{fold} :: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha\,list \rightarrow \beta \rightarrow \beta$.

3 Regular Expression Equivalence Framework

Regular expression (language) equivalence is usually reduced to (language) equivalence of automata. In principle our framework does the same, except that we construct the automata on the fly and replace the traditional transition table by computations on regular-expression-like objects. We start by relating regular expressions and automata.

Left quotients of a regular language L can be understood as states of a deterministic automaton $\mathscr{M}_L$ with the initial state L, the transition function $\mathscr{D}$, and the accepting states being those languages K for which $[] \in K$ holds. This automaton (restricted to reachable states) is finite and minimal by the Myhill-Nerode theorem.[1] The following locale captures this left-quotient-based view of an automaton:

[1] Note that the Myhill-Nerode relation $\approx_L$ can be defined as $v \approx_L w \longleftrightarrow \mathsf{fold}\ \mathscr{D}\ v\ L = \mathsf{fold}\ \mathscr{D}\ w\ L$. The quotient of $\mathscr{M}_L$ by this relation is isomorphic to $\mathscr{M}_L$; hence $\mathscr{M}_L$ is minimal.

locale $\mathit{rexpDA} =$
fixes $\iota :: \alpha\,\mathit{rexp} \to \sigma$ **and** $\mathsf{L} :: \sigma \to \alpha\,\mathit{lang}$ **and** $\delta :: \alpha \to \sigma \to \sigma$ **and** $\mathrm{o} :: \sigma \to \mathit{bool}$
assumes $\iota\mathsf{L}$: $\mathsf{L}\,(\iota\, r) = \mathscr{L}\, r$ **and** $\delta\mathsf{L}$: $\mathsf{L}\,(\delta\; a\; s) = \mathscr{D}\; a\;(\mathsf{L}\, s)$ **and** $\mathrm{o}\mathsf{L}$: $\mathrm{o}\, s \longleftrightarrow [] \in \mathsf{L}\, s$

The parameters ι and L formalize what “regular-expression-like” means: $\iota\, r$ embeds the regular expression r into a state of type σ, whereas L gives elements of σ a language semantics, which coincides with the language semantics of regular expressions by the assumption $\iota\mathsf{L}$. The function δ is the symbolic computation of left quotients on σ according to $\delta\mathsf{L}$. It can be regarded as the transition function of an automaton with states in σ and the initial state $\iota\, r$. Accepting states of this automaton are given by o.

Let us develop and verify some algorithms in the context of *rexpDA*. For a start, regular expression matching is easy to define

$$\mathsf{match}\; r\; w = \mathrm{o}\;(\mathsf{fold}\; \delta\; w\; (\iota\, r))$$

and prove correct: $\mathsf{match}\; r\; w \longleftrightarrow w \in \mathscr{L}\; r$.

Now we tackle the equivalence checker. We follow the well-known product automaton construction where language equivalence means $\mathrm{o}\; s_1 \longleftrightarrow \mathrm{o}\; s_2$ for all states (s_1, s_2) of the product automaton. Alternatively, one can view this procedure as the construction of a bisimulation relation between two automata: language equivalence and existence of a bisimulation coincide for deterministic automata [24]. The set of reachable states of an automaton can be obtained as the reflexive transitive closure of the start state under $\lambda p.\ \mathsf{map}\ (\lambda a.\ \delta\ a\ p)\ \mathit{as}$ where $\mathit{as} :: \alpha\, \mathit{list}$ is the alphabet.

We define a reflexive transitive closure operation

$$\mathsf{rtc} :: (\alpha \to \mathit{bool}) \to (\alpha \to \alpha\,\mathit{list}) \to \alpha \to (\alpha\,\mathit{list} \times \alpha\,\mathit{set})\,\mathit{option}$$

where type $\alpha\,\mathit{option}$ is the datatype $\mathsf{None} \mid \mathsf{Some}\,\alpha$. It is used to encode whether the closure is finite (Some is returned) or infinite (None is returned). The function rtc is defined using a while combinator and is executable (provided its arguments being executable); the result Some corresponds to a terminating computation [19]. The definition can be found in Isabelle/HOL’s library theory `While_Combinator` under its full name rtrancl_while. The parameters and result of $\mathsf{rtc}\; p\; \mathit{next}\; \mathit{start}$ have the following meaning: Predicate p is a test that stops the closure computation if an element not satisfying p is found; this is merely an optimization. Function *next* maps an element to a list of successors. Of course *start* is the start element. A result $\mathsf{Some}\;(\mathit{ws}, Z)$ means that the closure computation terminated with a worklist *ws* and a set of reachable elements Z. If *ws* is empty, Z is the set of all elements reachable from *start*; otherwise, the computation was stopped because an element not satisfying p was found. More precisely, we proved

$$\begin{array}{l}\mathsf{rtc}\; p\; \mathit{next}\; \mathit{start} = \mathsf{Some}\;(\mathit{ws}, Z) \implies \\ \textbf{if}\; \mathit{ws} = []\; \textbf{then}\; Z = R \wedge (\forall z \in Z.\ p\; z)\; \textbf{else}\; \neg p\; (\mathsf{hd}\; \mathit{ws}) \wedge \mathsf{hd}\; \mathit{ws} \in R\end{array} \qquad (1)$$

where $R = \{(x, y) \mid y \in \mathsf{set}\; (\mathit{next}\; x)\}^*$ “ $\{\mathit{start}\}$ and “ is infix relation application: r “ $\{x\} = \{y \mid (x, y) \in r\}$.

The state space of the product automaton is computed as follows:

$$\begin{array}{l}\mathsf{closure} :: \alpha\,\mathit{list} \to \sigma \times \sigma \to ((\sigma \times \sigma)\,\mathit{list} \times (\sigma \times \sigma)\,\mathit{set})\,\mathit{option} \\ \mathsf{closure}\; \mathit{as} = \mathsf{rtc}\; (\lambda (s, t).\ \mathrm{o}\, s \longleftrightarrow \mathrm{o}\, t)\; (\lambda (s, t).\ \mathsf{map}\; (\lambda a.\ (\delta\; a\; s, \delta\; a\; t))\; \mathit{as})\end{array}$$

The predicate $\lambda(s,t).\ \mathrm{o}\,s \longleftrightarrow \mathrm{o}\,t$ stops the computation as soon as a contradiction to language equality is found. The actual language equivalence checker merely needs to test if the worklist is empty at the end:

$$\begin{array}{l}
\mathsf{eqv} :: \alpha\,\mathit{rexp} \to \alpha\,\mathit{rexp} \to \mathit{bool}\\
\mathsf{eqv}\ r\ s = \mathbf{case}\ \mathsf{closure}\ (\mathsf{atoms}\ r\ @\ \mathsf{atoms}\ s)\ (\iota\ r, \iota\ s)\ \mathbf{of}\\
\qquad\qquad\qquad\ \ \mathsf{Some}\ ([], \_) \Rightarrow \mathsf{True}\\
\qquad\qquad\qquad\ \mid \_ \Rightarrow \mathsf{False}
\end{array}$$

The alphabet given to closure is the concatenation of the atoms in the two expressions.

Soundness of eqv is an easy consequence of the following property, which in turn follows from (1):

$$\begin{array}{l}
\mathsf{closure}\ (\mathsf{atoms}\ r\ @\ \mathsf{atoms}\ s)\ (\iota\ r, \iota\ s) = \mathsf{Some}\ (ws, Z) \implies\\
ws = [] \longleftrightarrow \mathsf{L}\ r = \mathsf{L}\ s
\end{array} \tag{2}$$

Theorem 1 (in *rexpDA*). eqv $r\ s \implies \mathscr{L}\,r = \mathscr{L}\,s$.

This is a partial correctness statement because it assumes that the call to closure in eqv returns Some, i.e. terminates.

Termination of closure needs finiteness of the underlying automaton. Therefore we extend *rexpDA* with an explicit assumption of finiteness:

$$\begin{array}{l}
\textbf{locale}\ \mathit{rexpDFA} = \mathit{rexpDA} +\\
\textbf{assumes}\,\mathit{fin}\text{: finite}\ \{\mathsf{fold}\ \delta\ w\ (\iota\ r) \mid w :: \alpha\,\mathit{list}\}
\end{array}$$

In this context the termination lemma for closure is an easy consequence of *fin* and the following termination property of rtc:

$$\mathsf{finite}\ (\{(x,y) \mid y \in \mathsf{set}\ (f\ x)\}^{*}\ \text{``}\ \{x\}) \implies \exists y.\ \mathsf{rtc}\ p\ f\ x = \mathsf{Some}\ y$$

Lemma 2 (in *rexpDFA*). closure $as\ (\iota\ r, \iota\ s) \neq$ None.

Together with (2) this implies completeness of eqv:

Theorem 3 (in *rexpDFA*). $\mathscr{L}\,r = \mathscr{L}\,s \implies$ eqv $r\ s$.

This is the end of all considerations about equivalence of regular expressions. The rest of the paper merely needs to focus on various methods for turning regular expressions into finite automata in the sense of *rexpDFA*.

Note that $\mathscr{M}_L$ defined above constitutes a first valid interpretation of *rexpDFA*. The proof of *fin* requires the Myhill-Nerode theorem.

$$\begin{array}{l}
\textbf{interpretation}\ \mathsf{M} : \mathit{rexpDFA}\ \textbf{where}\\
\quad \iota\ r \quad\ \ = \mathscr{L}\,r\\
\quad \delta\ a\ L = \mathscr{D}\ a\ L\\
\quad \mathrm{o}\,L \quad\ = [] \in L\\
\quad \mathsf{L}\,L \quad\ = L
\end{array}$$

This interpretation is not executable because neither its next-step function $\mathscr{D}$ (being based on infinite sets of words defined by a set comprehension) nor the equality on $\sigma = \alpha\,\mathit{lang}$ (which is needed for the closure computation) is executable.

4 Derivatives

In 1964, Brzozowski [7] showed how to compute left quotients syntactically—as derivatives of regular expressions. Derivatives have been rediscovered in proof assistants by Krauss and Nipkow [19] and Coquand and Siles [10]. Our first executable instantiations of the framework reuse infrastructure from earlier formalizations in Isabelle [19, 26].

A refinement of Brzozowski's approach, partial derivatives, was introduced by Antimirov [2] and formalized by Moreira *et al.* [21] in Coq and by Wu *et al.* [27] in Isabelle. Partial derivatives operate on finite sets of regular expressions. They can be viewed either as a nondeterministic automaton with regular expressions as states or as the corresponding deterministic automaton obtained by the subset construction.

In the following, we integrate the two notions in our framework and show how derivatives can be used to simulate partial derivatives without invoking sets explicitly.

4.1 Brzozowski's Derivatives

Given a letter c and a regular expression r, the (*Brzozowski*) *derivative* $\mathsf{der} :: \alpha \to \alpha\, rexp \to \alpha\, rexp$ of r w.r.t. a is defined by primitive recursion:

$$\begin{aligned}
&\mathsf{der}\ \_\ \mathbf{0} &&= \mathbf{0}\\
&\mathsf{der}\ \_\ \mathbf{1} &&= \mathbf{0}\\
&\mathsf{der}\ a\ (\mathsf{A}\ x) &&= \mathbf{if}\ x = a\ \mathbf{then}\ \mathbf{1}\ \mathbf{else}\ \mathbf{0}\\
&\mathsf{der}\ a\ (r + s) &&= \mathsf{der}\ a\ r + \mathsf{der}\ a\ s\\
&\mathsf{der}\ a\ (r \cdot s) &&= \mathbf{if}\ \mathsf{nullable}\ r\ \mathbf{then}\ (\mathsf{der}\ a\ r \cdot s) + \mathsf{der}\ a\ s\ \mathbf{else}\ \mathsf{der}\ a\ r \cdot s\\
&\mathsf{der}\ a\ (r^*) &&= \mathsf{der}\ a\ r \cdot r^*
\end{aligned}$$

It follows by induction on r that the language of the derivative $\mathsf{der}\ a\ r$ is exactly the left quotient $\mathscr{D}\ a\ (\mathscr{L}\ r)$. This property corresponds exactly to the assumption $\delta\mathsf{L}$ of the locale *rexpDA*. Hence it suggests the following interpretation:

interpretation *rexpDA* **where**
$\iota\ r = r \qquad \delta\ a\ r = \mathsf{der}\ a\ r \qquad \mathsf{o}\ r = \mathsf{nullable}\ r \qquad \mathsf{L}\ r = \mathscr{L}\ r$

Unfortunately, the sound equivalence checker that is produced by this interpretation is useless in practice, because it will rarely terminate. For example, the automaton constructed from the regular expression a^* is infinite, as all derivatives w.r.t. words a^n are distinct: $\mathsf{fold\ der}\ a^1\ a^* = \mathbf{1} \cdot a^*$; $\mathsf{fold\ der}\ a^{n+1}\ a^* = \mathbf{0} \cdot a^* + \mathsf{fold\ der}\ a^n\ a^*$.

Fortunately, Brzozowski showed that there are finitely many equivalence classes of derivatives modulo associativity, commutativity and idempotence (ACI) of the $+$ constructor. We prove that the number of distinct derivatives of r modulo ACI is finite: $\mathsf{finite}\ \{[\mathsf{fold\ der}\ w\ r]_\sim \mid w \in (\Sigma\ r)^*\}$ where $[r]_\sim = \{s \mid r \sim s\}$ denotes the equivalence class of r and the ACI equivalence $\sim$ is defined inductively as follows.

$$r + (s + t) \sim (r + s) + t \qquad r + s \sim s + r \qquad r + r \sim r$$

$$r \sim r \qquad \frac{r \sim s}{s \sim r} \qquad \frac{r \sim s \quad s \sim t}{r \sim t}$$

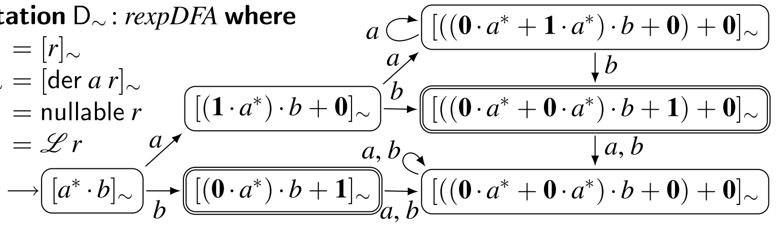
$$\frac{r_1 \sim s_1 \quad r_2 \sim s_2}{r_1 + r_2 \sim s_1 + s_2} \qquad \frac{r_1 \sim s_1 \quad r_2 \sim s_2}{r_1 \cdot r_2 \sim s_1 \cdot s_2} \qquad \frac{r \sim s}{r^* \sim s^*}$$

ACI-equivalent regular expressions $r \sim s$ have the same atoms and same languages, and their equivalence is preserved by the derivative: $\mathsf{der}\ b\ r \sim \mathsf{der}\ b\ s$ for all $b \in \Sigma\, r$.

This enables the following interpretation that operates on ACI equivalence classes. We obtain a first totally correct and complete equivalence checker $\mathsf{D}_\sim.\mathsf{eqv}$ in Isabelle/HOL.

interpretation $\mathsf{D}_\sim$: *rexpDFA* **where**

$$\begin{aligned}
\iota\, r &= [r]_\sim \\
\delta\, a\, [r]_\sim &= [\mathsf{der}\ a\ r]_\sim \\
\mathsf{o}\, [r]_\sim &= \mathsf{nullable}\ r \\
\mathsf{L}\, [r]_\sim &= \mathscr{L}\, r
\end{aligned}$$



a
$[((\mathbf{0}\cdot a^* + \mathbf{1}\cdot a^*)\cdot b + \mathbf{0}) + \mathbf{0}]_\sim$
a
b
b
$[(\mathbf{1}\cdot a^*)\cdot b + \mathbf{0}]_\sim$
$[((\mathbf{0}\cdot a^* + \mathbf{0}\cdot a^*)\cdot b + \mathbf{1}) + \mathbf{0}]_\sim$
a
a, b
a, b
$[a^*\cdot b]_\sim$
$[(\mathbf{0}\cdot a^*)\cdot b + \mathbf{1}]_\sim$
$[((\mathbf{0}\cdot a^* + \mathbf{0}\cdot a^*)\cdot b + \mathbf{0}) + \mathbf{0}]_\sim$
b
a, b

Fig. 1: Derivative automaton modulo ACI for $a^* \cdot b$

Technically, the formalization defines a quotient type [18] of "regular expressions modulo ACI" to represent equivalence classes and uses Lifting and Transfer [17] to lift operations on regular expressions to operations on equivalence classes. The above presentation of definitions of the locale parameters by "pattern matching" on equivalence classes resembles the code generated by Isabelle for quotients (a pseudo-constructor [14], $[\_]_\sim$, wraps a concrete representative r), rather than the actual definitions by Lifting.

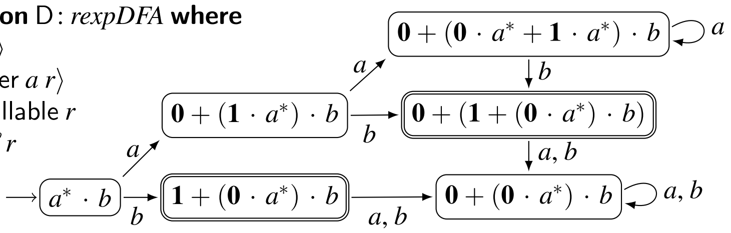
Since the equivalence checker must compare equivalence classes, the code generation for quotients requires an executable equality (i.e. a decision procedure for $\sim$-equivalence). We achieve this through an ACI normalization function $\langle\_\rangle$ that maps a regular expression r to a canonical representative of $[r]_\sim$ by sorting all summands w.r.t. an arbitrary fixed linear order $\preceq$ while removing duplicates. The definition of $\langle\_\rangle$ employs a smart (simplifying) constructor $\oplus$, whose equations are matched sequentially.

$$\begin{aligned}
\langle \mathbf{0} \rangle &= \mathbf{0} & (r + s) \oplus t &= r \oplus (s \oplus t) \\
\langle \mathbf{1} \rangle &= \mathbf{1} & r \oplus (s + t) &= \textbf{if } r = s \textbf{ then } s + t \\
\langle \mathsf{A}\, a \rangle &= \mathsf{A}\, a & &\ \textbf{else if } r \preceq s \textbf{ then } r + (s + t) \\
\langle r + s \rangle &= \langle r \rangle \oplus \langle s \rangle & &\ \qquad \textbf{else } s + (r \oplus t) \\
\langle r \cdot s \rangle &= \langle r \rangle \cdot \langle s \rangle & r \oplus s &= \textbf{if } r = s \textbf{ then } r \\
\langle r^* \rangle &= \langle r \rangle^* & &\ \textbf{else if } r \preceq s \textbf{ then } r + s \textbf{ else } s + r
\end{aligned}$$

We obtain an executable decision procedure for ACI equivalence: $r \sim s \longleftrightarrow \langle r \rangle = \langle s \rangle$. This makes $\mathsf{D}_\sim.\mathsf{eqv}$ executable, yielding verified code in different functional programming languages via Isabelle's code generator. Yet, the performance of the generated code is disappointing. Fig. 1 shows why: Derivations clutter concrete representatives with duplicated summands. Further derivation steps perform the same computation repeatedly and hence become increasingly expensive. This bottleneck is avoided by taking canonical ACI-normalized representatives as states yielding a second interpretation.

interpretation D : *rexpDFA* **where**

$$\begin{aligned}
\iota\, r &= \langle r \rangle \\
\delta\, a\, r &= \langle \mathsf{der}\ a\ r \rangle \\
\mathsf{o}\, r &= \mathsf{nullable}\ r \\
\mathsf{L}\, r &= \mathscr{L}\, r
\end{aligned}$$



$\mathbf{0} + (\mathbf{0}\cdot a^* + \mathbf{1}\cdot a^*)\cdot b$
a
a
b
$\mathbf{0} + (\mathbf{1}\cdot a^*)\cdot b$
$\mathbf{0} + (\mathbf{1} + (\mathbf{0}\cdot a^*)\cdot b)$
b
a
a, b
$a^*\cdot b$
$\mathbf{1} + (\mathbf{0}\cdot a^*)\cdot b$
$\mathbf{0} + (\mathbf{0}\cdot a^*)\cdot b$
a, b
b
a, b

Fig. 2: ACI-normalized derivative automaton for $a^* \cdot b$

A few points are worth mentioning here: First, D does not use the quotient type—it operates directly on canonical representatives and therefore can use structural equality for comparison (rather than $\sim$). Second, the interpretations $\mathsf{D}_\sim$ and D yield structurally the same automata, although with different labels. Fig. 2 shows the automaton produced by D for $a^* \cdot b$. This observation—which enables us to reuse the technically involved proof of $\mathsf{D}_\sim.\mathit{fin}$ to discharge $\mathsf{D}.\mathit{fin}$—relies crucially on our normalization function $\langle\_\rangle$ being idempotent and well-behaved w.r.t. derivatives:

Lemma 4. *We have* $\langle\langle r\rangle\rangle = \langle r\rangle$ *and* $\langle \mathsf{der}\ b\, \langle r\rangle\rangle = \langle \mathsf{der}\ b\ r\rangle$ *for all* $b \in \Sigma\ r$.

The automaton from Fig. 2 shows that the state labels still contain superfluous information, notably in the form of $\mathbf{0}$s and $\mathbf{1}$s. A coarser relation than $\sim$-equivalence, denoted $\approx$, adresses this concern. We omit the straightforward inductive definition of $\approx$, which cancels $\mathbf{0}$s and $\mathbf{1}$s where possible and takes the associativity of concatenation $\cdot$ into account. Coarseness ($[r]_\sim \subseteq [r]_\approx$) together with $\mathsf{D}_\sim.\mathit{fin}$ implies finiteness of equivalence classes of derivatives modulo $\approx$: $\mathsf{finite}\ \{[\mathsf{fold}\ \mathsf{der}\ w\ r]_\approx \mid w \in (\Sigma\ r)^*\}$.

As before, to avoid working with equivalence classes, we use a recursively defined $\approx$-normalization function $\langle\!\langle\_\rangle\!\rangle$ similar to $\langle\_\rangle$ (it corresponds to the *norm* function from the formalization by Krauss and Nipkow [19]). However, $\langle\!\langle\_\rangle\!\rangle$ (also $\approx$) is not well-behaved w.r.t. derivatives: for example, $\langle\!\langle \mathsf{der}\ a\ \langle\!\langle((a+\mathbf{1})\ \cdot\ (a\ \cdot\ a))\ \cdot\ b\rangle\!\rangle\rangle\!\rangle \neq \langle\!\langle \mathsf{der}\ a\ (((a+\mathbf{1})\ \cdot\ (a\ \cdot\ a))\ \cdot\ b)\rangle\!\rangle$. The normalization would need to take the distributivity of $\cdot$ over $+$ into account to prevent this disequality, but even with this addition a formal proof of well-behavedness seems difficult. Furthermore, our evaluation (Sect. 6) suggests that not too much energy should be invested in finding this proof. Thus, the following interpretation gives only a partial correctness result.

interpretation $\mathsf{N} : \mathit{rexpDA}$ **where**
$\iota\ r \quad = \langle\!\langle r\rangle\!\rangle$
$\delta\ a\ r = \langle\!\langle \mathsf{der}\ a\ r\rangle\!\rangle$
$\mathsf{o}\ r \quad = \mathsf{nullable}\ r$
$\mathsf{L}\ r \quad = \mathscr{L}\ r$

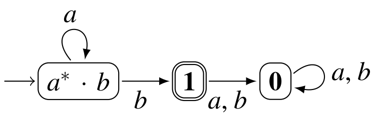


Fig. 3: Normalized derivative automaton for $a^*\ \cdot\ b$

In practice, we did not find an input for which N would construct an infinite automaton. For the example $a^*\ \cdot\ b$ it even yields the minimal automaton shown in Fig. 3.

4.2 Partial Derivatives

Partial derivatives split certain $+$-constructors into sets of regular expressions, thus capturing ACI equivalence directly in the data structure. The automaton construction for a regular expression r starts with the singleton set $\{r\}$. More precisely, partial derivatives $\mathsf{pder} :: \alpha \to \alpha\,\mathit{rexp} \to (\alpha\,\mathit{rexp})\,\mathit{set}$ are defined recursively as follows:

$$
\begin{aligned}
\mathsf{pder}\ \_\ \mathbf{0} &= \{\} \\
\mathsf{pder}\ \_\ \mathbf{1} &= \{\} \\
\mathsf{pder}\ a\ (\mathsf{A}\ x) &= \mathbf{if}\ x = a\ \mathbf{then}\ \{\mathbf{1}\}\ \mathbf{else}\ \{\} \\
\mathsf{pder}\ a\ (r + s) &= \mathsf{pder}\ a\ r \cup \mathsf{pder}\ a\ s \\
\mathsf{pder}\ a\ (r\ \cdot\ s) &= \mathbf{if}\ \mathsf{nullable}\ r\ \mathbf{then}\ (\mathsf{pder}\ a\ r \odot s) \cup \mathsf{pder}\ a\ s\ \mathbf{else}\ \mathsf{pder}\ a\ r \odot s \\
\mathsf{pder}\ a\ (r^*) &= \mathsf{pder}\ a\ r \odot r^*
\end{aligned}
$$

Above, $R \odot s$ is used as a shorthand notation for $\{r \cdot s \mid r \in R\}$. The definition yields the characteristic property of partial derivatives by induction on r:

$$\mathscr{D}\; a\; (\mathscr{L}\, r) = \bigcup_{s \in \mathsf{pder}\; a\; r} \mathscr{L}\, s$$

Following this characteristic property, we can interpret the locale *rexpDFA*. The automaton constructed by P for our running example is shown in Fig. 4.

interpretation P: *rexpDFA* **where**

$$\begin{array}{lcl} \iota\, r & = & \{r\} \\ \delta\; a\; R & = & \bigcup_{r \in R} \mathsf{pder}\; a\; r \\ \mathrm{o}\, R & = & \exists r \in R.\ \mathsf{nullable}\; r \\ \mathsf{L}\, R & = & \bigcup_{r \in R} \mathscr{L}\, r \end{array}$$

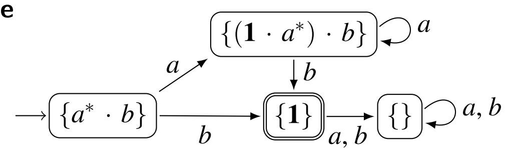


Fig. 4: Partial derivative automaton for $a^* \cdot b$

The assumptions of *rexpDA* (inherited by *rexpDFA*) are easy to discharge. Just as for Brzozowski derivatives, only the proof of finiteness of the reachable state space P.*fin* poses a challenge. We were able to reuse the proof by Wu *et al.* [27] who show finiteness when proving one direction of the Myhill-Nerode theorem. Compared with the proof of D.*fin*, the formal reasoning about partial derivatives appears to be more succinct.

There is a direct connection between pder and der that seems not to have been covered in the literature. It is best expressed in terms of a recursive function pset :: $\alpha\, rexp \to (\alpha\, rexp)\, set$ that translates derivatives to partial derivatives: $\mathsf{pset}\; (\mathsf{der}\; a\; r) = \mathsf{pder}\; a\; r$.

$$\begin{array}{lcl@{\qquad}lcl} \mathsf{pset}\; \mathbf{0} & = & \{\} & \mathsf{pset}\; (r + s) & = & \mathsf{pset}\; r \cup \mathsf{pset}\; s \\ \mathsf{pset}\; \mathbf{1} & = & \{\mathbf{1}\} & \mathsf{pset}\; (r \cdot s) & = & \mathsf{pset}\; r \odot s \\ \mathsf{pset}\; (\mathsf{A}\; x) & = & \{\mathsf{A}\; x\} & \mathsf{pset}\; (r^*) & = & \{r^*\} \end{array}$$

A finite set R of regular expressions can be represented uniquely by a single regular expression $\sum R$, a sum ordered w.r.t. $\preceq$. Hence, we have $\sum \mathsf{pset}\; (\mathsf{der}\; a\; r) = \sum \mathsf{pder}\; a\; r$, meaning that we can devise a normalization function $\langle\!\langle r \rangle\!\rangle = \sum \mathsf{pset}\; r$ that allows us to simulate partial derivatives while operating on plain regular expressions. Alternatively, $\langle\!\langle \_ \rangle\!\rangle$ can be defined using smart constructors (with sequentially matched equations):

$$\begin{array}{lcl@{\qquad}lcl} \langle\!\langle \mathbf{0} \rangle\!\rangle & = & \mathbf{0} & \mathbf{0} \boxplus r & = & r \\ \langle\!\langle \mathbf{1} \rangle\!\rangle & = & \mathbf{1} & r \boxplus \mathbf{0} & = & r \\ \langle\!\langle \mathsf{A}\; a \rangle\!\rangle & = & \mathsf{A}\; a & (r + s) \boxplus t & = & r \boxplus (s \boxplus t) \\ \langle\!\langle r + s \rangle\!\rangle & = & \langle\!\langle r \rangle\!\rangle \boxplus \langle\!\langle s \rangle\!\rangle & r \boxplus (s + t) & = & \textbf{if}\; r = s\; \textbf{then}\; s + t \\ \langle\!\langle r \cdot s \rangle\!\rangle & = & \langle\!\langle r \rangle\!\rangle \boxdot s & & & \textbf{else if}\; r \preceq s\; \textbf{then}\; r + (s + t) \\ \langle\!\langle r^* \rangle\!\rangle & = & r^* & & & \quad \textbf{else}\; s + (r \boxplus t) \\ \mathbf{0} \boxdot r & = & \mathbf{0} & r \boxplus s & = & \textbf{if}\; r = s\; \textbf{then}\; r \\ (r + s) \boxdot t & = & (r \boxdot s) \boxplus (s \boxdot t) & & & \textbf{else if}\; r \preceq s\; \textbf{then}\; r + s \\ r \boxdot s & = & s \cdot t & & & \quad \textbf{else}\; s + r \end{array}$$

This definition allows to contrast the implicit quotienting performed by partial derivatives with the qoutienting modulo ACI equivalence ($\sim$). They turn out to be incomparable: $\langle\!\langle \_ \rangle\!\rangle$ does not simplify the second argument of concatenation $\cdot$ and the argument of iteration *, but erases $\mathbf{0}$s and uses left distributivity.

Finally, we obtain a last derivative-based interpretation using the characteristic property $\langle\!\langle \mathsf{der}\; b\; r \rangle\!\rangle = \sum(\mathsf{pder}\; b\; r)$ and P.*fin* to discharge the finiteness assumption *fin*.

interpretation PD: *rexpDFA* **where**

$$
\begin{aligned}
\iota\, r &= \langle\!\langle r \rangle\!\rangle\\
\delta\, a\, r &= \langle\!\langle \mathsf{der}\ a\ r \rangle\!\rangle\\
\mathrm{o}\, r &= \mathsf{nullable}\ r\\
\mathsf{L}\, r &= \mathscr{L}\, r
\end{aligned}
$$

Whenever P yields an automaton for r with states labeled with finite sets of regular expressions X_i, PD constructs structurally the same automaton for r labeled with $\sum X_i$.

5 Marked Regular Expressions

One of the oldest methods for converting a regular expression into an automaton is based on the idea of identifying the states of the automaton with positions in the regular expression. Both McNaughton and Yamada [20] and Glushkov [13] mark the atoms in a regular expression with numbers to identify positions uniquely. In this section, we formalize two recent reincarnations of this approach due to Fischer *et al.* [11] and Asperti [3]. They are based on the realization that in a functional programming setting, it is most convenient to represent positions in a regular expression by marking some of its atoms. First we define an infrastructure for working with marked regular expressions. Then we define and relate both reincarnations in terms of this infrastructure.

Marked regular expressions are formalized by the following type synonym (where the value True denotes a marked atom)

$$\alpha\ \mathit{mrexp} = (\mathit{bool} \times \alpha)\ \mathit{rexp}$$

We convert easily between *rexp* and *mrexp* with the help of map_rexp, the map function on regular expressions:

$$
\begin{aligned}
&\mathsf{strip} = \mathsf{map\_rexp}\ \mathsf{snd}\\
&\mathsf{emtpy\_mrexp} = \mathsf{map\_rexp}\ (\lambda r.\ (\mathsf{False}, r))
\end{aligned}
$$

The language $\mathscr{L}_m :: \alpha\ \mathit{mrexp} \to \alpha\ \mathit{lang}$ of a marked regular expression is the set of words that start at some marked atom:

$$
\begin{aligned}
\mathscr{L}_m\ \mathbf{0} &= \{\}\\
\mathscr{L}_m\ \mathbf{1} &= \{\}\\
\mathscr{L}_m\ (\mathsf{A}\ (m, a)) &= \mathbf{if}\ m\ \mathbf{then}\ \{[a]\}\ \mathbf{else}\ \{\}\\
\mathscr{L}_m\ (r + s) &= \mathscr{L}_m\ r \cup \mathscr{L}_m\ s\\
\mathscr{L}_m\ (r \cdot s) &= (\mathscr{L}_m\ r \cdot \mathscr{L}(\mathsf{strip}\ s)) \cup \mathscr{L}_m\ s\\
\mathscr{L}_m\ (r^*) &= \mathscr{L}_m\ r \cdot \mathscr{L}(\mathsf{strip}\ r)^*
\end{aligned}
$$

The function $\mathsf{final} :: \alpha\ \mathit{mrexp} \to \mathit{bool}$ tests if some atom at the "end" of a given regular expression is marked:

$$
\begin{aligned}
\mathsf{final}\ \mathbf{0} &= \mathsf{False}\\
\mathsf{final}\ \mathbf{1} &= \mathsf{False}\\
\mathsf{final}\ (\mathsf{A}\ (m, a)) &= m\\
\mathsf{final}\ (r + s) &= (\mathsf{final}\ r \lor \mathsf{final}\ s)\\
\mathsf{final}\ (r \cdot s) &= (\mathsf{final}\ s \lor \mathsf{nullable}\ s \land \mathsf{final}\ r)\\
\mathsf{final}\ (r^*) &= \mathsf{final}\ r
\end{aligned}
$$

Marks are moved around a regular expression by two operations. The function $\mathsf{read}\ a\ r$ unmarks all atoms in r except a:

$$\begin{aligned}&\mathsf{read} :: \alpha \to \alpha\ \mathit{mrexp} \to \alpha\ \mathit{mrexp}\\ &\mathsf{read}\ a = \mathsf{map\_rexp}\ (\lambda\,(m, x).\ (m \wedge a = x, x))\end{aligned}$$

Its characteristic lemma is that it restricts $\mathscr{L}_m\ r$ to words whose head is a:

$$\mathscr{L}_m\ (\mathsf{read}\ a\ r) = \{w \in \mathscr{L}_m\ r \mid w \neq [] \wedge \mathsf{hd}\ w = a\}$$

The function $\mathsf{follow}\ m\ r$ moves all marks in r to the "next" atom, much like an ε-closure; the mark m is pushed in from the left:

$$\begin{aligned}&\mathsf{follow} :: \mathit{bool} \to \alpha\ \mathit{mrexp} \to \alpha\ \mathit{mrexp}\\ &\mathsf{follow}\ m\ \mathbf{0} &&= \mathbf{0}\\ &\mathsf{follow}\ m\ \mathbf{1} &&= \mathbf{1}\\ &\mathsf{follow}\ m\ (\mathsf{A}\ (\_, a)) &&= \mathsf{A}\ (m, a)\\ &\mathsf{follow}\ m\ (r + s) &&= \mathsf{follow}\ m\ r + \mathsf{follow}\ m\ s\\ &\mathsf{follow}\ m\ (r \cdot s) &&= \mathsf{follow}\ m\ r\ \cdot\ \mathsf{follow}\ (\mathsf{final}\ r \vee m \wedge \mathsf{nullable}\ r)\ s\\ &\mathsf{follow}\ m\ (r^*) &&= (\mathsf{follow}\ (\mathsf{final}\ r \vee m)\ r)^*\end{aligned}$$

The characteristic lemma about follow shows that the marks are moved forward, thereby chopping off the first letter (in the generated language), and that the parameter m indicates whether every "first" atom should be marked:

$$\mathscr{L}_m\ (\mathsf{follow}\ m\ r) = \{\mathsf{tl}\ w \mid w \in \mathscr{L}_m\ r\} \cup (\mathbf{if}\ m\ \mathbf{then}\ \mathscr{L}(\mathsf{strip}\ r)\ \mathbf{else}\ \{\}) - \{[]\}$$

5.1 Mark After Atom

In the work of McNaughton-Yamada-Glushkov, the mark indicates which atom has just been read, i.e. the mark is located "after" the atom. Therefore the initial state is special because nothing has been read yet. Thus we express the states of the automaton as a pair of a boolean (True means that nothing has been read yet) and a marked regular expression. The boolean can be viewed as a mark in front of the automaton. (Alternatively, one could work with an explicit start symbol in front of the regular expression.) We interpret the locale *rexpDFA* as follows:

$$\begin{aligned}&\mathbf{interpretation}\ \mathsf{A}: \mathit{rexpDFA}\ \mathbf{where}\\ &\quad \iota\ r &&= (\mathsf{True}, \mathsf{emtpy\_mrexp}\ r)\\ &\quad \delta\ a\ (m, r) &&= (\mathsf{False}, \mathsf{read}\ a\ (\mathsf{follow}\ m\ r))\\ &\quad \mathrm{o}\ (m, r) &&= (\mathsf{final}\ r \vee m \wedge \mathsf{nullable}\ r)\\ &\quad \mathsf{L}\ (m, r) &&= \mathscr{L}_m(\mathsf{follow}\ m\ r) \cup (\mathbf{if}\ \mathrm{o}\ (m, r)\ \mathbf{then}\ \{[]\}\ \mathbf{else}\ \{\})\end{aligned}$$

The definition of δ expresses that we first build the ε-closure starting from the marked atoms (via follow) and then read the next atom. With the characteristic lemmas about read and follow (and a few auxiliary lemmas), the locale assumptions are easily proved. This yields our first version of automata based on marked regular expressions.

Finiteness of the reachable part of the state space is proved via the lemma

$$\mathsf{fold}\ \delta\ w\ (\iota\ r) \in \{\mathsf{True}, \mathsf{False}\} \times \mathsf{mrexps}\ r$$

where $\mathsf{mrexps} :: \alpha\, \mathit{rexp} \to (\alpha\, \mathit{mrexp})\, \mathit{set}$ maps a regular expression to the finite set of all its marked variants, i.e. $\mathsf{mrexps}\ r = \{r' \mid \mathsf{strip}\ r' = r\}$; its actual recursive definition is straightforward and omitted.

Now we take a closer look at the work of Fischer *et al.* [11], which inspired the preceding formalization. They present a number of (not formally verified) matching algorithms on marked regular expressions in Haskell that follow McNaughton-Yamada-Glushkov. This is their basic transition function:

$$
\begin{aligned}
&\mathsf{shift} :: \mathit{bool} \to \alpha\ \mathit{mrexp} \to \alpha \to \alpha\ \mathit{mrexp}\\
&\mathsf{shift}\ \_\ \mathbf{0}\ \_ &&= \mathbf{0}\\
&\mathsf{shift}\ \_\ \mathbf{1}\ \_ &&= \mathbf{1}\\
&\mathsf{shift}\ m\ (\mathsf{A}\ (\_, x))\ c &&= \mathsf{A}\ (m \wedge (x = c), x)\\
&\mathsf{shift}\ m\ (r + s)\ c &&= \mathsf{shift}\ m\ r\ c + \mathsf{shift}\ m\ s\ c\\
&\mathsf{shift}\ m\ (r \cdot s)\ c &&= \mathsf{shift}\ m\ r\ c\ \cdot\ \mathsf{shift}\ (\mathsf{final}\ r \vee m \wedge \mathsf{nullable}\ r)\ s\ c\\
&\mathsf{shift}\ m\ (r^*)\ c &&= (\mathsf{shift}\ (\mathsf{final}\ r \vee m)\ r\ c)^*
\end{aligned}
$$

A simple induction proves that their shift is our δ:

$$\mathsf{shift}\ m\ r\ x = \mathsf{read}\ x\ (\mathsf{follow}\ m\ r)$$

Thus we have verified their shift function. Fischer *et al.* optimize shift further, which is still quadratic due to the calls of the recursive functions final and nullable. They simply cache the values of final and nullable at all nodes of a regular expression by adding additional fields to each constructor. We have verified this optimization step as well, yielding another interpretation A_2 (omitted here).

5.2 Mark Before Atom

Instead of imagining the mark to be after an atom, it can also be viewed to be in front of it, i.e. it marks possible next atoms. This is somewhat dual to the McNaughton-Yamada-Glushkov construction. It leads to the following interpretation of the *rexpDA* locale:

$$
\begin{aligned}
&\textbf{interpretation}\ \mathsf{B} : \mathit{rexpDFA}\ \textbf{where}\\
&\quad \iota\ r &&= (\mathsf{follow\ True}\ (\mathsf{emtpy\_mrexp}\ r), \mathsf{nullable}\ r)\\
&\quad \delta\ a\ (r, m) &&= \textbf{let}\ r' = \mathsf{read}\ a\ r\ \textbf{in}\ (\mathsf{follow\ False}\ r', \mathsf{final}\ r')\\
&\quad \mathsf{o}\ (r, m) &&= m\\
&\quad \mathsf{L}\ (r, m) &&= \mathscr{L}_m\ r \cup (\textbf{if}\ m\ \textbf{then}\ \{[]\}\ \textbf{else}\ \{\})
\end{aligned}
$$

The definition of δ expresses that we first read an atom and then build the ε-closure. The assumptions of *rexpDA* and *rexpDFA* are proved easily just like in the previous interpretation with marked regular expressions.

The interesting point is that this happens to be the algorithm formalized by Asperti [3]. Although he says that he has formalized McNaughton-Yamada, he actually formalized the dual algorithm. This is not easy to see because Asperti's formalization is considerably more involved than ours, with many auxiliary functions. Strictly speaking, his algorithm is a variation of ours that produces the same automata. The complete proof of this fact can be found elsewhere [16]. Because of the size of Asperti's formalization, there is not enough space here to give the detailed equivalence proof. However, we can take a step towards his formulation and merge follow and read into one function $\mathsf{move} :: \alpha \to \alpha\ \mathit{mrexp} \to \mathit{bool} \to \alpha\ \mathit{mrexp}$, the analogue of his homonymous function:

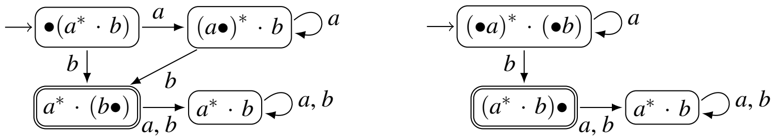


Fig. 5: Marked regular expression automata (A left, B right) for $a^* \cdot b$

$$
\begin{aligned}
&\mathsf{move}\ \_\ \mathbf{0}\ \_ &&= \mathbf{0}\\
&\mathsf{move}\ \_\ \mathbf{1}\ \_ &&= \mathbf{1}\\
&\mathsf{move}\ c\ (\mathsf{A}\ (\_, x))\ m &&= \mathsf{A}\ (m, x)\\
&\mathsf{move}\ c\ (r + s)\ m &&= \mathsf{move}\ c\ r\ m + \mathsf{move}\ c\ s\ m\\
&\mathsf{move}\ c\ (r \cdot s)\ m &&= \mathsf{move}\ c\ r\ m\ \cdot\ \mathsf{move}\ c\ s\ (\mathsf{final1}\ r\ c \lor m \land \mathsf{nullable}\ r)\\
&\mathsf{move}\ c\ (r^*)\ m &&= (\mathsf{move}\ c\ r\ (\mathsf{final1}\ r\ c \lor m))^*
\end{aligned}
$$

where $\mathsf{final1}$ is an auxiliary recursive function (not shown here) with the characteristic property that $\mathsf{final1}\ r\ c = \mathsf{final}\ (\mathsf{read}\ c\ r)$. A simple induction proves that move combines follow and read as in δ:

$$\mathsf{move}\ c\ r\ m = \mathsf{follow}\ m\ (\mathsf{read}\ c\ r)$$

The function move has quadratic complexity for the same reason as shift. Unfortunately, it cannot be made linear with the same ease as for shift. The problem is that we need to cache the value of $\mathsf{final1}\ r\ c$ in the previous step, before we know c. We solve this by caching the set of all letters c that make $\mathsf{final1}\ r\ c$ true. In the worst case, the whole alphabet must be stored in certain inner nodes. However, for an alphabet of fixed size this guarantees linear time complexity. This optimization constitutes a last interpretation B_2.

Even for a fixed alphabet, Asperti's move has quadratic complexity when faced with a tower of stars: each recursive call of move can trigger a call of a function eclose, which has linear complexity. Asperti aimed for compact proofs, not maximal efficiency.

5.3 Comparison

The two constructions may look similar, but they do not produce isomorphic automata. Considering our running example, we display the mark by a "$\bullet$" before or after the atom. The two resulting automata are shown in Fig. 5. There are special states that cannot be denoted by marking atoms only: $\bullet r$ in A's automaton is the completely unmarked regular expression that is the initial state and $r\bullet$ in B's automaton is a final state.

It turns out that the "before" automaton is a homomorphic image of the "after" automaton. To verify this we specify the homomorphism $\varphi\,(m, r) = (\mathsf{follow}\ m\ r, \mathsf{A.o}\ (m, r))$ and prove that it preserves initial states and commutes with the transition function:

$$\varphi\,(\mathsf{A}.\iota\ r) = \mathsf{B}.\iota\ r \qquad \varphi\,(\mathsf{A}.\delta\ a\ s) = \mathsf{B}.\delta\ a\ (\varphi\ s) \qquad \varphi\,(\mathsf{fold}\ \mathsf{A}.\delta\ w\ s) = \mathsf{fold}\ \mathsf{B}.\delta\ w\ (\varphi\ s)$$

A direct consequence is that Asperti's "before" construction always generates automata with at most as many states as the McNaughton-Yamada-Glushkov construction. Formally, in the context of locale *rexpDA* we have defined an executable computation of the reachable state space $\{\mathsf{fold}\ \delta\ w\ (\iota\ r) \mid w \in (\mathsf{set}\ as)^*\}$ of the automaton:

$$\mathsf{reachable}\ as\ r = \mathsf{snd}\ (\mathsf{the}\ (\mathsf{rtc}\ (\lambda\_.\ \mathsf{True})\ (\lambda s.\ \mathsf{map}\ (\lambda a.\ \delta\ a\ s)\ as)\ (\iota\ r)))$$

where r is the initial regular expression, as is the alphabet, and the $(\mathsf{Some}\ x) = x$.

Theorem 5. $|\mathsf{B.reachable}\ as\ r| \leq |\mathsf{A.reachable}\ as\ r|$ *where* $|\_|$ *is the cardinality of a set.*

In early drafts of this paper, we only conjectured the above statement and unsuccessfully tried to refute it with Isabelle's Quickcheck facility [8]. Later, Helmut Seidl has communicated an informal proof using the above homomorphism to us.

Let us abbreviate the statement of Thm. 5 to $n_b \leq n_a$. One may think that n_a is only slightly larger than n_b, but it seems that n_b and n_a are more than a constant summand apart: for a two-element alphabet Quickcheck could refute $n_a \leq n_b + k$ even for $k = 100$.

6 Empirical Comparison

We compare the efficiency w.r.t. both matching and deciding equivalence of the Standard ML code generated from eight described interpretations: $\sim$-normalized derivatives (D), $\approx$-normalized derivatives (N), partial derivatives (P), derivatives simulating partial derivatives (PD), mark "after" atom (A), mark "after" atom with caching (A_2), mark "before" atom (B), and mark "before" atom with caching (B_2). The interpretation using the quotient type for derivatives ($\mathsf{D}_\sim$) is not in this list, as it is clearly superseded by D. The results of the evaluation, performed on an Intel Core i7-2760QM machine with 8 GB of RAM, are shown in Fig. 6. Solid lines depict the four derivative-based algorithms. Dashed lines are used for the algorithms based on marked regular expressions.

The first two tests, MATCH-R and MATCH-L, measure the time required to match the word a^n against the regular expression $(a + \mathbf{1})^n \cdot a^n$—a standard benchmark also used by Fischer *et al.* [11]. The difference between the two tests is the definition of r^n. MATCH-R defines it as the n-fold concatenation associated to the right: $r^4 = r \cdot (r \cdot (r \cdot r))$, whereas MATCH-L associates to the left: $r^4 = ((r \cdot r) \cdot r) \cdot r$. In both tests, marked regular expressions outperform derivatives by far. The normalization performed by the derivative-based approaches (required to obtain a finite number of states for the equivalence check) decelerates the computation of the next state. Marked regular expressions benefit from a fast next state computation. The test MATCH-L exhibits the quadratic nature of the unoptimized matchers A and B (their curves are almost identical and therefore hard to distinguish in Fig. 6). In contrast, A_2 and B_2 perform equally well in both tests, A_2 being approximately 1.5 times faster due to lighter cache annotations.

The next test goes back to Antimirov [1]: We measure the time (with a timeout of ten seconds) for proving the equivalence of a^* and $(a^0 + \ldots + a^{n-1}) \cdot (a^n)^*$. Again two tests, EQ-R and EQ-L, distinguish the associativity of concatenation in r^n. Here, the derivative-based equivalence checkers (except for D) perform better then the ones based on marked regular expressions. In particular, both version of partial derivatives, P and PD, outperform N—since this example was crafted by Antimirov to demonstrate the strength of partial derivatives, this is not wholly unexpected. Comparing EQ-R and EQ-L, the associativity barely influences the runtime.

Finally, to avoid bias towards a particular algorithm, we have devised the randomized test EQ-RND. There we measure the average time (with a timeout of ten seconds) to prove the equivalence of r with itself for 100 randomly generated expressions with n inner nodes ($+$, $\cdot$, or *). Proving $r \equiv r$ is of course a trivial task, but our algorithms do not

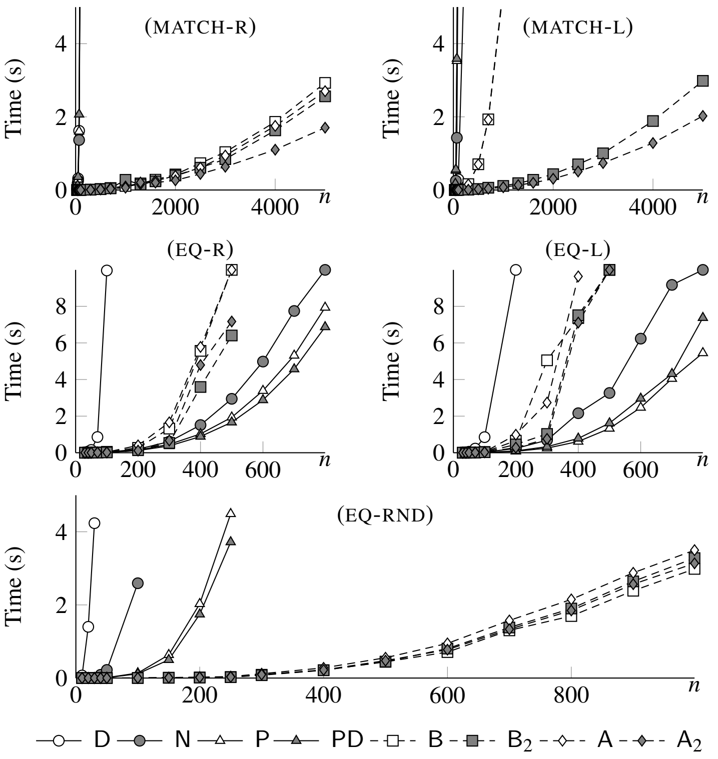


Fig. 6: Evaluation results

stop the exploration when the state of the product automaton is a pair of two equal states. This optimization, which is a must for any practical algorithm, is the first step towards the rewarding usage of bisimulation up to equivalence (or even up to congruence) [5]. Without any such optimization, the task of proving $r \equiv r$ amounts to enumerating all derivatives of r, which is exactly what we want to compare. To generate random regular expression we use the infrastructure of SpecCheck [25]—a Quickcheck clone for Isabelle/ML. For computing the average, a timeout counts as 10 second (although the actual computation would likely have taken longer)—an approximation that skews the curves to converge to the margin of 10 seconds. We stopped measuring a method for increasing n when the average approached 5 seconds.

The results of EQ-RND are summarized as follows: $\mathsf{D} \gg \mathsf{N} \gg \mathsf{P}, \mathsf{PD} \gg \mathsf{A}, \mathsf{A}_2, \mathsf{B}, \mathsf{B}_2$, where $X \gg Y$ means that Y is an order of magnitude faster than X. The algorithm P is noticeably slower than PD—avoiding sets reduces the overhead. Among A, A_2, B, B_2, Asperti's unoptimized algorithm B performs best by a narrow margin. Regular expressions where the caching overhead pays off are rare and therefore not visible in the randomized test results. The same holds for expressions where B produces much smaller automata than A (e.g. the counterexample to $n_a \leq n_b + 100$ from Subsect. 5.3).

Our evaluation shows that A_2 is the best choice for matching. For equivalence checking, the winner is not as clear cut: B (especially when applied to normalized input to avoid quadratic runtime without caching) and PD seem to be the best choices.

7 Extensions

Brzozowski's derivatives are easily extendable to regular expressions intersection and negation—indeed Brzozowski performed the extension right from the start [7]. The number of such extended derivatives is still finite when quotiented modulo ACI.

We [26] have recently further extended derivatives to regular expressions extended with projection, obtaining verified decision procedures for the equivalence of those extended regular expressions and for monadic second-order logics over finite words. The closure computation and its correctness proof follow Krauss and Nipkow [19].

Extending partial derivatives with intersection and negation is more involved [9]. An additional layer of sets must be used for intersections, i.e. the states of our automaton would then be sets of sets of regular expressions. In Sect. 6, we have seen that already one layer of sets incurs some overhead. Hence, the view on partial derivatives as derivatives followed by some normalization is expected to be even more profitable for the extension. The extension of partial derivatives with projection is an easy exercise.

It is unclear how to extend marked regular expressions to handle negation and intersection. The number of possible markings for a regular expression of alphabetic width n is 2^n. However, there exist regular expressions of alphabetic width n using intersection, whose minimal automata have 2^{2^n} states [12].

8 Conclusion

We have shown that all the previously published verified decision procedures for equivalence of regular expressions that operate on regular expressions directly can all be expressed as instances of a generic automaton-inspired framework. The correctness proofs decompose into a generic part that is proved once and for all in the framework and a few specific properties that need to be proved for each instance. The framework caters for a meaningful comparison of the performance of the various instances. Marked regular expressions are superior on average but partial derivatives can outperform them in specific cases. The Isabelle theories are available online [23].

Acknowledgment. We thank Andrea Asperti and Sebastian Fischer for commenting on fine points of their work and Helmut Seidl for contributing an informal proof of Thm. 5. Jasmin Blanchette, Andrei Popescu and three anonymous reviewers helped to improve the presentation through numerous suggestions. The second author is supported by the doctorate program 1480 (PUMA) of the Deutsche Forschungsgemeinschaft (DFG).

References

1. Antimirov, V.: Partial derivatives of regular expressions and finite automata constructions. In: Mayr, E.W., Puech, C. (eds.) STACS 95. LNCS, vol. 900, pp. 455–466. Springer (1995)
2. Antimirov, V.: Partial derivatives of regular expressions and finite automaton constructions. Theor. Comput. Sci. 155(2), 291–319 (1996)
3. Asperti, A.: A compact proof of decidability for regular expression equivalence. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 283–298. Springer (2012)
4. Ballarin, C.: Interpretation of locales in Isabelle: Theories and proof contexts. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS, vol. 4108, pp. 31–43. Springer (2006)

5. Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: Giacobazzi, R., Cousot, R. (eds.) POPL 2013. pp. 457–468. ACM (2013)
6. Braibant, T., Pous, D.: An efficient Coq tactic for deciding Kleene algebras. In: Kaufmann, M., Paulson, L. (eds.) ITP 2010. LNCS, vol. 6172, pp. 163–178. Springer (2010)
7. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM 11(4), 481–494 (1964)
8. Bulwahn, L.: The new Quickcheck for Isabelle: Random, exhaustive and symbolic testing under one roof. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 92–108. Springer (2012)
9. Caron, P., Champarnaud, J.M., Mignot, L.: Partial derivatives of an extended regular expression. In: Dediu, A.H., Inenaga, S., Martín-Vide, C. (eds.) LATA 2011. LNCS, vol. 6638, pp. 179–191. Springer (2011)
10. Coquand, T., Siles, V.: A decision procedure for regular expression equivalence in type theory. In: Jouannaud, J.P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 119–134. Springer (2011)
11. Fischer, S., Huch, F., Wilke, T.: A play on regular expressions: functional pearl. In: Hudak, P., Weirich, S. (eds.) ICFP 2010. pp. 357–368. ACM (2010)
12. Gelade, W., Neven, F.: Succinctness of the complement and intersection of regular expressions. ACM Trans. Comput. Log. 13(1), 4:1–19 (2012)
13. Glushkov, V.M.: The abstract theory of automata. Russian Math. Surveys 16, 1–53 (1961)
14. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 100–115. Springer (2013)
15. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer (2010)
16. Haslbeck, M.: Verified Decision Procedures for the Equivalence of Regular Expressions. B.Sc. thesis, Department of Informatics, Technische Universität München (2013)
17. Hufmann, B., Kunčar, O.: Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 131–146. Springer (2013)
18. Kaliszyk, C., Urban, C.: Quotients revisited for Isabelle/HOL. In: Chu, W.C., Wong, W.E., Palakal, M.J., Hung, C.C. (eds.) SAC 2011. pp. 1639–1644. ACM (2011)
19. Krauss, A., Nipkow, T.: Proof pearl: Regular expression equivalence and relation algebra. J. Automated Reasoning 49, 95–106 (2012), published online March 2011
20. McNaughton, R., Yamada, H.: Regular expressions and finite state graphs for automata. IRE Trans. on Electronic Comput EC-9, 38–47 (1960)
21. Moreira, N., Pereira, D., de Sousa, S.M.: Deciding regular expressions (in-)equivalence in Coq. In: Kahl, W., Griffin, T. (eds.) RAMiCS 2012. LNCS, vol. 7560, pp. 98–113. Springer (2012)
22. Nipkow, T., Klein, G.: Concrete Semantics. A Proof Assistant Approach. Springer (to appear), `http://www.in.tum.de/~nipkow/Concrete-Semantics`
23. Nipkow, T., Traytel, D.: Regular expression equivalence. Archive of Formal Proofs (2014), `http://afp.sf.net/entries/Regex_Equivalence.shtml`, Formal proof development
24. Rutten, J.J.M.M.: Automata and coinduction (an exercise in coalgebra). In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 194–218. Springer (1998)
25. Schaffroth, N.: A Specification-based Testing Tool for Isabelle's ML Environment. B.Sc. thesis, Department of Informatics, Technische Universität München (2013)
26. Traytel, D., Nipkow, T.: Verified decision procedures for MSO on words based on derivatives of regular expressions. In: Morrisett, G., Uustalu, T. (eds.) ICFP 2013. pp. 3–12 (2013)
27. Wu, C., Zhang, X., Urban, C.: A formalisation of the Myhill-Nerode theorem based on regular expressions. J. Automated Reasoning, 52, 451–480 (2014)