Noname manuscript No.
(will be inserted by the editor)

Proof Pearl: Regular Expression Equivalence and Relation Algebra

Alexander Krauss and Tobias Nipkow

the date of receipt and acceptance should be inserted later

Abstract We describe and verify an elegant equivalence checker for regular expressions. It works by constructing a bisimulation relation between (derivatives of) regular expressions. By mapping regular expressions to binary relations, an automatic and complete proof method for (in)equalities of binary relations over union, composition and (reflexive) transitive closure is obtained. The verification is carried out in the theorem prover Isabelle/HOL, yielding a practically useful decision procedure.

1 Introduction

The equational theory of regular expressions is convenient for reasoning about binary relations. For example, the theories of Thiemann and Sternagel [12] contain the lemma

$$S \circ (S \circ S^* \circ R^* \cup R^*) \subseteq S \circ S^* \circ R^*,$$

followed by a long-winded low-level proof. Here, R and S are binary relations, $\circ$ is relation composition and * is the reflexive transitive closure. However, this is just an inequality of regular expressions (interpreted over relations instead of regular languages), which is a decidable theory.

The purpose of this proof pearl is to a verify a simple decision procedure for regular expression equivalences, and to show how to reduce equations between binary relations to equations over languages. Put together, this yields an automatic proof procedure for relation algebra equalities (and inequalities, since $A \subseteq B$ is equivalent to $A \cup B = B$) that proves statements like the one above automatically. We formalized and verified the procedure in the theorem prover Isabelle/HOL.

The background of this research is the desire for completely formalized and verified decision procedures that do not rely on any unverified components or informal pen-and-paper proofs. This level of perfection has become the standard in the interactive theorem proving community but can be non-trivial to achieve. Therefore, our contribution is not the presentation of a new algorithm or proof but the concise, elegant, and fully mechanized presentation of a known algorithm (though probably not yet as widely known as it deserves) and its direct application to interactive theorem proving.

The standard equivalence test for two regular expressions goes like this:

Institut für Informatik, Technische Universität München

1. Convert both regular expressions into finite automata.
2. Make the automata deterministic, and possibly minimize them, and then compare them for language equality.

This proves that the two expressions denote the same regular language. Due to Kozen's theorem [7], the equality must then be a theorem in all Kleene Algebras, since regular languages are the initial model of Kleene Algebras. Thus, to apply the procedure to relations, one can

3. Formalize Kozen's theorem.
4. Prove that relations are Kleene Algebras.

This is the path followed by Braibant and Pous [4], who also motivate their work by proofs in relation algebra. However, formalizing Kozen's theorem is not easy—the proof amounts to replaying the well-known automaton constructions in an algebraic setting, using matrices. Moreover, the automata theory needed for steps 1 and 2 does not come for free either.

This inspired us to look for a more direct approach to achieve the same goal. We decided to follow the coalgebraic approach pioneered by Rutten [11], which involves derivatives and does not need automata or matrices. We deliberately do not discuss extensions of this approach, e.g., to Kleene algebras with tests [8] or generalized regular expressions [2], which would merely be a distraction from our main goal.

Unlike Braibant and Pous, we do not cover the general case of arbitrary Kleene algebras, since the main practical application are proofs about relations. For this application, our 750 line development [9] offers a low-cost and elegant alternative to their 19 000 line development at http://sardes.inrialpes.fr/~braibant/atbr/.

1.1 Equivalence Checking with Derivatives

In 1964, Brzozowski [5] showed how to convert a regular expression directly into a deterministic automaton whose states are derivatives of the intitial expression. The *derivative* $D_a(r)$ of a regular expression r w.r.t. a symbol a is a regular expression that describes the language of all words w for which aw is in the language of r. That is, $D_a(r)$ is what is left of r after an initial a, which corresponds to an a-transition from a "state" r to a "next state" $D_a(r)$. The derivative of a regular expression is easy to compute recursively (see Sect. 3).

This yields a procedure for building up the automaton transition by transition until we reach a closure. This happens after finitely many steps because Brzozowski showed that modulo associativity, commutativity and idempotence of $+$ there are only finitely many iterated derivatives reachable from any given r.

Owens *et al.* [10] have implemented Brzozowski's algorithm in functional languages. They write

> Regular expression derivatives have been lost in the sands of time

because standard textbooks do not refer to them. However, researchers do, for example Rutten [11]. He informally describes a neat algorithm for deciding equivalence of regular expressions r and s: incrementally construct the relation of all $(D_w(r), D_w(s))$ between the two state spaces, where $D_w(r)$ are iterated derivatives of r, and w is extended symbol by symbol. This process enumerates all pairs of states that must behave the same w.r.t. acceptance of input words, provided r and s are equivalent. The process must terminate by the above finiteness argument. It must either terminate with a bisimulation relation (in which case r and s are equivalent) or it must find a pair (r', s') where one of them is a final state while the other is not.

1.2 Overview

For the sake of minimality we concentrate on proving equivalence (When did you last want to *prove* $r \not\equiv s$?) and verify partial correctness. Proving termination and completeness belongs in the realm of meta-theory and is not required to obtain actual equivalence proofs — it merely lets you sleep better.

We first introduce the basics of regular languages and expressions (§§2–3). After introducing some normalization functions and the derivative operation itself (§4), we define bisimulations (§5) and an algorithm that computes them (§6). §7 combines the parts to an equivalence checker. The bridge from regular languages to relations is made in §8, which finally introduces our new proof method *regexp*.

All definitions and lemmas in this paper are direct renderings of their formal counterparts in Isabelle/HOL. We sometimes give informal proof sketches to help intuition; the full proofs can be found online [9].

2 Languages

Languages are sets of words, and words are lists of atoms (characters). The empty list is denoted by $[\,]$, adding an element x to the front of a list xs is written $x{:}xs$, and concatenation of two lists is written xs @ ys. Thus, $[\,]$ replaces the symbol ϵ that textbooks often use for the empty word. Lists denoting words are written using the variables v, w instead of xs, ys.

On languages, the operations of concatenation (@@), power (A^n) and Kleene star are defined in the usual manner:

$$A \mathbin{@@} B = \{v \mathbin{@} w \mid v \in A \wedge w \in B\}$$

$$A^{0} = \{[\,]\}$$
$$A^{n+1} = A \mathbin{@@} A^{n}$$

$$A^{*} = (\bigcup n\ A^{n})$$

In addition we have the derivative operation w.r.t. a single character a:

$$\mathit{deriv}\ a\ L = \{w \mid a{:}w \in L\}$$

Derivation obeys the following lemmas:

$$\mathit{deriv}\ a\ \emptyset = \emptyset$$
$$\mathit{deriv}\ a\ \{[\,]\} = \emptyset$$
$$\mathit{deriv}\ a\ \{[b]\} = (\textit{if}\ a = b\ \textit{then}\ \{[\,]\}\ \textit{else}\ \emptyset)$$
$$\mathit{deriv}\ a\ (A \cup B) = \mathit{deriv}\ a\ A \cup \mathit{deriv}\ a\ B$$
$$\mathit{deriv}\ a\ (A \mathbin{@@} B) =$$
$$(\textit{if}\ [\,] \in A\ \textit{then}\ \mathit{deriv}\ a\ A \mathbin{@@} B \cup \mathit{deriv}\ a\ B\ \textit{else}\ \mathit{deriv}\ a\ A \mathbin{@@} B)$$
$$\mathit{deriv}\ a\ (A^{*}) = \mathit{deriv}\ a\ A \mathbin{@@} A^{*}$$

Equality of two languages can be shown by coinduction. The following lemma is the key to the correctess proof of our decision procedure.

Lemma 1 *Let $\sim$ be a binary relation between languages such that*
1. *for all A and B, $A \sim B \Longrightarrow [\,] \in A \longleftrightarrow [\,] \in B$, and*
2. *for all A and B and x, $A \sim B \Longrightarrow$ deriv x $A \sim$ deriv x B.*

Then $A \sim B$ implies $A = B$.

Proof By symmetry, it is enough to show that for all A and B, $A \sim B$ and $w \in A$ imply $w \in B$. We proceed by induction on w. For $w = [\,]$, we have $w \in B$ using property 1. For $w = a{:}w'$, we have $w' \in$ *deriv a A* and by induction hypothesis $w' \in$ *deriv a B* (since *deriv a A* $\sim$ *deriv a B* due to property 2.). Thus, $w \in B$.

A relation $\sim$ with the properties 1. and 2. above is called a *bisimulation*.

Thus, to prove that two languages are equal, we must show that they are contained in a bisimulation. Our algorithm will construct such a bisimulation explicitly, as a list of pairs of regular expressions.

3 Regular Expressions

Regular expressions are defined as a recursive datatype α *rexp* where α is the underlying alphabet. The constructors are $\ll\cdot\gg$ (for single characters), $+$ (for sum), $\cdot$ (for concatenation), * (for Kleene star), and $\mathbf{0}$ and $\mathbf{1}$ as additive and multiplicative identities.

The language of a regular expression is defined in the standard way, as is the set of atoms in an expression:

$$
\begin{array}{ll}
L(\mathbf{0}) = \emptyset & L(r + s) = L(r) \cup L(s) \\
L(\mathbf{1}) = \{[\,]\} & L(r \cdot s) = L(r) \mathbin{@@} L(s) \\
L(\ll a\gg) = \{[a]\} & L(r^*) = (L(r))^*
\end{array}
$$

$$
\begin{array}{ll}
\mathit{atoms}\ \mathbf{0} = \emptyset & \mathit{atoms}\ (r + s) = \mathit{atoms}\ r \cup \mathit{atoms}\ s \\
\mathit{atoms}\ \mathbf{1} = \emptyset & \mathit{atoms}\ (r \cdot s) = \mathit{atoms}\ r \cup \mathit{atoms}\ s \\
\mathit{atoms}\ \ll a\gg = \{a\} & \mathit{atoms}\ (r^*) = \mathit{atoms}\ r
\end{array}
$$

We will also need a function *final*, which determines if the language of an expression contains the empty word, i.e., corresponds to a final state.

$$
\begin{array}{ll}
\mathit{final}\ \mathbf{0} \longleftrightarrow \mathit{False} & \mathit{final}\ (r + s) \longleftrightarrow \mathit{final}\ r \vee \mathit{final}\ s \\
\mathit{final}\ \mathbf{1} \longleftrightarrow \mathit{True} & \mathit{final}\ (r \cdot s) \longleftrightarrow \mathit{final}\ r \wedge \mathit{final}\ s \\
\mathit{final}\ \ll a\gg \longleftrightarrow \mathit{False} & \mathit{final}\ (r^*) \longleftrightarrow \mathit{True}
\end{array}
$$

By induction we obtain the characteristic property *final* $r \longleftrightarrow [\,] \in L(r)$.

4 Normal Forms and Derivatives

Before we define the derivative of a regular expression, we first introduce a normalization function, which rewrites expressions with a number of equational laws.

Most importantly, the normalization function must identify expressions that are equivalent modulo associativity, commutativity and idempotence (ACI) of $+$, which will ensure termination of the closure computation (due to Brzozowski, see §1.1). However, we also include a few other simplifications, such the laws for **0** and **1** and associativity of $\cdot$. Having more simplifications is a good thing, since it can only make the resulting automaton smaller.

Our normalization function maps the constructors $+$ and $\cdot$ to functions $\oplus$ and $\odot$, which combine already normalized subterms into a normalized term.

$$\begin{array}{ll}
norm\ \mathbf{0} = \mathbf{0} & norm\ (r + s) = norm\ r \oplus norm\ s \\
norm\ \mathbf{1} = \mathbf{1} & norm\ (r \cdot s) = norm\ r \odot norm\ s \\
norm\ \ll a \gg\ = \ \ll a \gg & norm\ (r^*) = (norm\ r)^*
\end{array}$$

Function $\odot$ does the obvious thing with **0** and **1** and parenthesizes nested concatenations to the right. Function $\oplus$ also parenthesizes expressions to the right, i.e., turns them into a list, but in addition sorts the list w.r.t some total order $\preceq$, and eliminates duplicates and **0**s:

$$\begin{array}{ll}
\mathbf{0} \odot \_ = \mathbf{0} & \mathbf{0} \oplus r = r \\
\_ \odot \mathbf{0} = \mathbf{0} & r \oplus \mathbf{0} = r \\
\mathbf{1} \odot r = r & (r + s) \oplus t = r \oplus (s \oplus t) \\
r \odot \mathbf{1} = r & r \oplus (s + t) = \\
(r \cdot s) \odot t = r \cdot s \odot t & (\textit{if}\ r = s\ \textit{then}\ s + t \\
r \odot s = r \cdot s & \ \textit{else if}\ r \preceq s\ \textit{then}\ r + (s + t) \\
 & \qquad \textit{else}\ s + r \oplus t) \\
 & r \oplus s = \\
 & (\textit{if}\ r = s\ \textit{then}\ r \\
 & \ \textit{else if}\ r \preceq s\ \textit{then}\ r + s\ \textit{else}\ s + r)
\end{array}$$

The relation $\preceq$ can be some arbitrary total order on regular expressions. Our concrete definition (omitted) works as follows. Expressions with different constructors at the root are compared according to some arbitrary fixed total order of the constructors. Expressions with the same constructor at the root are compared according to the lexicographic order of the arguments. Atoms are assumed to be totally ordered. More precisely, $\preceq$ is restricted to regular expressions over natural numbers.

It is easy to prove the following two lemmas by induction, with the help of similar properties for the auxiliary functions $\oplus$ and $\odot$, which we omit here.

Lemma 2 $L(norm\ r) = L(r)$

Lemma 3 $atoms\ (norm\ r) \subseteq atoms\ r$

We now proceed to define the derivative $D_a(r)$ of a regular expression r. The following definition reflects the properties of language derivatives from §2, but we use $\oplus$ and $\odot$ instead of $+$ and $\cdot$, thus ensuring that normal forms are preserved.

$$\begin{array}{l}
D_a(\mathbf{0}) = \mathbf{0} \\
D_a(\mathbf{1}) = \mathbf{0} \\
D_a(\ll b \gg) = (\textit{if}\ a = b\ \textit{then}\ \mathbf{1}\ \textit{else}\ \mathbf{0}) \\
D_a(r + s) = D_a(r) \oplus D_a(s) \\
D_a(r \cdot s) = (\textit{if final}\ r\ \textit{then}\ D_a(r) \odot s \oplus D_a(s)\ \textit{else}\ D_a(r) \odot s) \\
D_a(r^*) = D_a(r) \odot r^*
\end{array}$$

By induction we show the characteristic property $L(D_a(r)) = deriv\ a\ (L(r))$.

5 Bisimulations Between Regular Expressions

The predicate *is-bisimulation* checks if a list of pairs of regular expressions forms a bisimulation, and if all expressions in *ps* contain only atoms from the list *as*:

$$\begin{array}{l} \textit{is-bisimulation as ps} \longleftrightarrow \\ (\forall\,(r,\ s)\in\textit{set ps}. \\ \quad \textit{atoms } r \cup \textit{atoms } s \subseteq \textit{set as} \wedge \\ \quad (\textit{final } r \longleftrightarrow \textit{final } s) \wedge (\forall\, a\in\textit{set as}.\ (D_a(r),\ D_a(s)) \in \textit{set ps})) \end{array}$$

Function *set* converts a list into a set.

Compared to the relation $\sim$ in Lemma 1, this finitary notion of bisimulation as a list of pairs is executable. Thus, *ps* and *as* act as a certificate for the equivalence of two expressions. The following is a consequence of Lemma 1:

Lemma 4 *If is-bisimulation as ps and* $(r,\ s) \in$ *set ps then* $L(r) = L(s)$.

6 Computing the Bisimulation Closure

Strictly speaking, Lemma 4 is all we need to generate formal proofs of regular expression equivalences. We could generate the list of pairs *ps* using some untrusted piece of code, and then run the predicate *is-bisimulation* to check that it really is a bisimulation.

In other settings, where checking a certificate is much easier than producing it, this approach can be a great advantage. But in our case, checking that a relation is a bisimulation is about as expensive as generating it, so we will spend a little bit of effort on verifying the closure algorithm that produces the bisimulation. Then, no dynamic check is needed.

6.1 The *while-option* Combinator

We want to define and reason about a closure computation without having to prove its termination. For such situations, Isabelle's library defines a variant of the well-known *while* combinator, which is called *while-option*. It takes a test $b :: \alpha \Rightarrow \textit{bool}$, a function $c :: \alpha \Rightarrow \alpha$, and a "state" $s :: \alpha$, and obeys the recursion equation

$$\textit{while-option}\ b\ c\ s = (\mathsf{if}\ b\ s\ \mathsf{then}\ \textit{while-option}\ b\ c\ (c\ s)\ \mathsf{else}\ \textit{Some}\ s)$$

This equation is executable, but the execution diverges if $b\ s$, $b\ (c\ s)$, $b\ (c\ (c\ s))$, ... are all true. In this case, the result of the function is specified as *None* (this is the logical specification; the actual execution will loop infinitely).

The definition of the combinator shall not concern us here. We merely use the recursion equation, together with an invariant-based proof rule for the case where the loop terminates (thus returning *Some*):

Lemma 5 (While-rule) *If*

1. *P is an invariant* (*for all s*, $P\ s \Longrightarrow b\ s \Longrightarrow P\ (c\ s)$),
2. *the execution terminates* (*while-option* $b\ c\ s = \textit{Some}\ t$), *and*
3. *the invariant holds initially* ($P\ s$)

then $P\ t$.

6.2 The closure computation

We use a standard iterative algorithm to compute the bisimulation. It uses a list ws of expression pairs that are not yet processed (the worklist), and another list of pairs ps, which works as an accumulator. In each step we move a pair (r, s) from ws over to ps, and for every atom a we add the pair of successors $(D_a(r), D_a(s))$ to ws provided it does not yet occur in ps or ws. The process terminates if either ws becomes empty (then ps holds the bisimulation) or when we encounter a pair where $\mathit{final}\ r \longleftrightarrow \mathit{final}\ s$ is false (then there is no bisimulation containing the initial ws). We define this process with the help of *while-option*:

$$\mathit{closure}\ \mathit{as} = \mathit{while\text{-}option}\ \mathit{test}\ (\mathit{step}\ \mathit{as})$$

Parameter as is the list of all atoms. The process continues while *test* is true:

$$\mathit{test}\ (\mathit{ws}, \mathit{ps}) \longleftrightarrow (\mathsf{case}\ \mathit{ws}\ \mathsf{of}\ [\,] \Rightarrow \mathit{False} \mid (p, q){:}x \Rightarrow \mathit{final}\ p \longleftrightarrow \mathit{final}\ q)$$

Each step is defined like this, where *hd* and *tl* take the head and tail of a list.

```
$\mathit{step}\ \mathit{as}\ (\mathit{ws}, \mathit{ps}) =$
$(\mathsf{let}\ (r, s) = \mathit{hd}\ \mathit{ws};$
     $\mathit{ps}' = (r, s){:}\mathit{ps};$
     $\mathit{succs} = \mathit{map}\ (\lambda a.\ (D_a(r), D_a(s)))\ \mathit{as};$
     $\mathit{new} = \mathit{filter}\ (\lambda p.\ p \notin \mathit{set}\ \mathit{ps}' \cup \mathit{set}\ \mathit{ws})\ \mathit{succs}$
 $\mathsf{in}\ (\mathit{new}\ @\ \mathit{tl}\ \mathit{ws}, \mathit{ps}'))$
```

To show that successful runs of *closure* find a bisimulation, we need to specify the invariant. The predicate *pre-bisim* characterizes when (ws, ps) contains the original pair (r, s), uses only atoms from as, and is on the way to a bisimulation:

```
$\mathit{pre\text{-}bisim}\ \mathit{as}\ r\ s\ (\mathit{ws}, \mathit{ps}) \longleftrightarrow$
$(r, s) \in \mathit{set}\ \mathit{ws} \cup \mathit{set}\ \mathit{ps}\ \wedge$
$(\forall (r, s){\in}\mathit{set}\ \mathit{ws} \cup \mathit{set}\ \mathit{ps}.\ \mathit{atoms}\ r \cup \mathit{atoms}\ s \subseteq \mathit{set}\ \mathit{as})\ \wedge$
$(\forall (r, s){\in}\mathit{set}\ \mathit{ps}.$
    $(\mathit{final}\ r \longleftrightarrow \mathit{final}\ s) \wedge (\forall a{\in}\mathit{set}\ \mathit{as}.\ (D_a(r), D_a(s)) \in \mathit{set}\ \mathit{ps} \cup \mathit{set}\ \mathit{ws}))$
```

It is easy to show that $\mathit{pre\text{-}bisim}\ \mathit{as}\ r\ s$ is an invariant of $\mathit{step}\ \mathit{as}$. Using the while rule and because $\mathit{pre\text{-}bisim}\ \mathit{as}\ r\ s\ ([\,], \mathit{ps})$ implies the premises of Lemma 4, we obtain

Lemma 6 *If* $\mathit{closure}\ \mathit{as}\ ([(r, s)], [\,]) = \mathit{Some}\ ([\,], \mathit{ps})$ *and* $\mathit{atoms}\ r \cup \mathit{atoms}\ s \subseteq \mathit{set}\ \mathit{as}$ *then* $L(r) = L(s)$.

7 The Equivalence Checker

The overall equivalence checker *check-eqv* takes two expressions, normalizes them, feeds them together with their atoms into *closure*, and checks that *closure* has terminated with an empty worklist:

```
$\mathit{check\text{-}eqv}\ r\ s =$
$(\mathsf{case}\ \mathit{closure}\ (\mathit{add\text{-}atoms}\ r\ (\mathit{add\text{-}atoms}\ s\ [\,]))\ ([(\mathit{norm}\ r, \mathit{norm}\ s)], [\,])\ \mathsf{of}$
  $\mathit{Some}([\,],\_) \Rightarrow \mathit{True} \mid \_ \Rightarrow \mathit{False})$
```

$$\mathit{add\text{-}atoms}\ \mathbf{0}\ \mathit{as} = \mathit{as}$$
$$\mathit{add\text{-}atoms}\ \mathbf{1}\ \mathit{as} = \mathit{as}$$

$$\begin{aligned}
&\textit{add-atoms} \ll a \gg \textit{as} = (\textsf{if } a \in \textit{set as } \textsf{then } \textit{as } \textsf{else } a{:}\textit{as}) \\
&\textit{add-atoms } (r + s) \textit{ as} = \textit{add-atoms } s \ (\textit{add-atoms } r \textit{ as}) \\
&\textit{add-atoms } (r \cdot s) \textit{ as} = \textit{add-atoms } s \ (\textit{add-atoms } r \textit{ as}) \\
&\textit{add-atoms } (r^*) \textit{ as} = \textit{add-atoms } r \textit{ as}
\end{aligned}$$

The final soundness result is a simple consequence from Lemma 6 and the definition of *check-eqv*.

Lemma 7 $\textit{check-eqv } r\ s \Longrightarrow L(r) = L(s)$

We can now reduce any proof obligation $L(r) = L(s)$ to *check-eqv r s*. Since the latter is executable, the proof is reduced to a mere computation. This computation can either be performed by Isabelle's simplifier, which is fast enough for small examples, or Isabelle can compile it to ML [1,6] where it can be evaluated efficiently, and the result can be re-imported into the logic afterwards.

8 Proving Equalities in Relation Algebra

Equivalences between regular expressions remain valid when interpreted over binary relations. This interpretation is relative to a *valuation* v of type $\alpha \Rightarrow (\beta \times \beta)$ *set*, which maps atoms to (homogenous) binary relations. Then, the relation of a regular expression is defined as follows, where *Id* is the identity relation and $\circ$ is relation composition.

$$\begin{aligned}
&\mathcal{R}_v(\mathbf{0}) = \emptyset && \mathcal{R}_v(r + s) = \mathcal{R}_v(r) \cup \mathcal{R}_v(s) \\
&\mathcal{R}_v(\mathbf{1}) = \textit{Id} && \mathcal{R}_v(r \cdot s) = \mathcal{R}_v(r) \circ \mathcal{R}_v(s) \\
&\mathcal{R}_v(\ll a \gg) = v\ a && \mathcal{R}_v(r^*) = (\mathcal{R}_v(r))^*
\end{aligned}$$

We define an auxiliary function $\mathcal{W}$, which interprets a single word as a relation:

$$\mathcal{W}_v([]) = \textit{Id} \qquad \mathcal{W}_v(a{:}\textit{as}) = v\ a \circ \mathcal{W}_v(\textit{as})$$

The following crucial lemma, proved by induction, connects $\mathcal{R}_v(r)$, $L(r)$, and $\mathcal{W}_v(r)$.

Lemma 8 $\mathcal{R}_v(r) = (\bigcup_{w \in L(r)} \mathcal{W}_v(w))$

As an immediate corollary, we obtain that the relation interpretation preserves equalities that hold over languages:

Lemma 9 $L(r) = L(s) \Longrightarrow \mathcal{R}_v(r) = \mathcal{R}_v(s)$

Now we use our procedure to prove equality of binary relations constructed by union, composition and reflexive transitive closure by a generic process often called *reflection* and originally developed by Boyer and Moore [3]. For example, the goal $(R^* \circ S)^* = (R \cup S)^*$, where R and S are relations, is proved in two steps. First it is transformed into its canonical interpretation under $\mathcal{R}$, resulting in the goal $\mathcal{R}_v(\textit{lhs}) = \mathcal{R}_v(\textit{rhs})$, where *lhs* and *rhs* are the expressions $(\ll 0 \gg^* \cdot \ll 1 \gg)^*$ and $(\ll 0 \gg + \ll 1 \gg)^*$ (over the alphabet $\{0,\ 1\} \subseteq \mathbb{N}$), and v is the valuation that maps 0 to R and 1 to S. This first step is automatic, given the definition of $\mathcal{R}$. The second step applies Lemmas 9 and 7 to obtain the goal *check-eqv lhs rhs* and solves this by evaluation.

Our Isabelle theories offer a proof method *regexp* that combines the two steps, and in addition turns inequalities into equalities (using $A \subseteq B \longleftrightarrow A \cup B = B$) and eliminates transitive closure (using $R^+ = R \circ R^*$). This means that the lengthy proof of our motivating example is now fully automated:

lemma $S \circ (S \circ S^* \circ R^* \cup R^*) \subseteq S \circ S^* \circ R^*$
by *regexp*

9 Discussion

Termination and Completeness Although we have argued why we did not not verify termination and completeness of our algorithm, we should still outline why they hold. Termination of the *closure* procedure follows along the lines of Brzozowski's original proof [5]. We define the set $\mathcal{D}(r)$ inductively by the following rules.

$$\begin{array}{l} \mathit{norm}\ r \in \mathcal{D}(r) \\ s \in \mathcal{D}(r) \Longrightarrow D_a(s) \in \mathcal{D}(r) \end{array}$$

By induction on the structure of r one can prove that $\mathcal{D}(r)$ is always finite. The interesting part of the proof is the case for r^*. Here we observe that each element of $\mathcal{D}(r^*)$ must be a sum of expressions of the form $s \cdot r^*$, where $s \in \mathcal{D}(r)$. Since $\mathcal{D}(r)$ is finite by induction hypothesis, $\mathcal{D}(r^*)$ must be finite, too. Since there are only finitely many different iterated derivatives, the closure algorithm must terminate. The main tedium in the formalization of this proof would be to show that *norm* really maps ACI-equivalent expressions to the same normal form, which is not necessary for the soundness proof.

Completeness of the procedure is now easy to see. If the *closure* algorithm terminates with a non-empty worklist, then it implicitly found a counterexample: Each pair of expressions in the worklist corresponds to a pair of reachable states for some word w. Thus, if a pair is found where $\mathit{final}\ r \longleftrightarrow \mathit{final}\ s$ does not hold, then w is in one of the languages but not the other. Extending the algorithm to actually output that counterexample is a simple exercise.

How about the completeness of the transition between relations and languages from §8? In fact there is a corner case where it is not complete. If the relations are over a type with only finitely many elements, then there also exist only finitely many different relations, and the beautiful structure of regular expressions is destroyed. In particular, if there are only n different relations, then the equation $R^0 \cup R^1 \cup \cdots \cup R^n = R^*$ holds (one of the relations on the left hand side must occur twice, so there is a cycle), but it is not valid over languages. However, this is really just nitpicking, as for the interesting case—relations over an infinite type— the procedure is complete. If an equality holds for all relations then we can interpret each atom a as the relation $v\ a = \bigcup w\ \{(f\ (a{:}w),\ f\ w)\}$, where f is some injective mapping from words to elements, which must exist since the type is infinite. Then for each expression r, $w \in L(r) \longleftrightarrow (f\ w,\ f\ [\,]) \in \mathcal{R}_v(r)$ and thus $\mathcal{R}_v(r) = \mathcal{R}_v(s) \Longrightarrow L(r) = L(s)$.

Heterogeneous Relations Braibant and Pous also handle equalities of heterogeneous relations, whose the domain and range may belong to different types. Our development does not cover this case directly.

However, instead of solving the problem with heterogeneous relations over types $\sigma_1, \ldots, \sigma_n$, we can simply solve the corresponding homogeneous one over the sum type $\tau = \sigma_1 + \cdots + \sigma_n$. In the resulting theorem we instantiate the variables with terms of the form $L_i \circ A \circ R_j$, where $A :: (\sigma_i \times \sigma_j)\ \mathit{set}$ and where $L_i :: (\tau \times \sigma_i)\ \mathit{set}$, $R_j :: (\sigma_j \times \tau)\ \mathit{set}$ are the left- and right-embeddings. The embeddings can then be simplified away to obtain the heterogeneous result.

This process is easy to automate and shows that a theory of homogeneous relations is sufficient for practical purposes. However, we have not implemented this step, since we have not found an application for it yet.

Performance In the large range of algorithms that turn regular expressions into automata, Brzozowski's procedure is on the elegant side, not the efficient one. The same is probably true for our equivalence checker. But our goal is to have a tool that quickly solves goals about relations. These goals arise in interactive proofs and aren't very large, so there is little point in optimizing performance. Equations involving a handful of variables, as they typically appear in proofs are solved instantly. We have not experimented much with scaling up, but it seems that for larger expressions, the automata-based implementation by Braibant and Pous clearly outperforms our checker.

10 Conclusion

We presented a decision procedure for regular expression equivalence and used it to prove equations in relation algebra in Isabelle/HOL. What makes this a proof pearl is the surprising ease with which this is achieved. Compared to the development by Braibant and Pous, we cut corners in three places.

1. Our equivalence checker using derivatives is "more algebraic" than automata and can be formalized with less conceptual and notational overhead.
2. We only formalized the soundness proof, which is sufficient for providing the proof method *regexp*. While the termination and completeness properties are not hard to formalize, they would not make the method more useful in practice.
3. Instead of general Kleene algebras, we concentrate on the main use case: binary relations. Thus, the full generality of Kozen's theorem is not needed, and we can use the simple reasoning of §8.

In a nutshell, the succinctness of our development is the result of the chosen formalization and of concentrating on the essentials.

Acknowledgement We thank Georg Struth for pointing us to the work of Jan Rutten.

References

1. Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *Lect. Notes in Comp. Sci.*, pages 24–40. Springer-Verlag, 2000.
2. M. M. Bonsangue, G. Caltais, E. Goriac, D. Lucanu, J. J.M.M. Rutten, and A. Silva. A decision procedure for bisimilarity of generalized regular expressions. In *Brazilian Symposium on Formal Methods (SBMF 2010)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2010.
3. Robert S. Boyer and J Strother Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In R. Boyer and J Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, 1981.
4. Thomas Braibant and Damien Pous. An efficient Coq tactic for deciding Kleene algebras. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *Lect. Notes in Comp. Sci.*, pages 163–178. Springer-Verlag, 2010.
5. Janusz Brzozowski. Derivatives of regular expressions. *J. ACM*, 11:481–494, 1964.
6. Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *Lect. Notes in Comp. Sci.*, pages 103–117. Springer-Verlag, 2010.

7. Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110:366–390, 1994.
8. Dexter Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical Report http://hdl.handle.net/1813/10173, Computing and Information Science, Cornell University, March 2008.
9. Alexander Krauss and Tobias Nipkow. Regular sets and expressions. In G. Klein, T. Nipkow, and L.C. Paulson, editors, *The Archive of Formal Proofs*. http://afp.sf.net/entries/Regular-Sets.shtml, May 2010. Formal proof development.
10. Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. of Functional Programming*, 19:173–190, 2009.
11. Jan J.M.M. Rutten. Automata and coinduction (an exercise in coalgebra). In D. Sangiorgi and R. de Simone, editors, *Concurrency Theory (CONCUR'98)*, volume 1466 of *Lect. Notes in Comp. Sci.*, pages 194–218. Springer-Verlag, 1998.
12. René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lect. Notes in Comp. Sci.*, pages 452–468. Springer-Verlag, 2009.