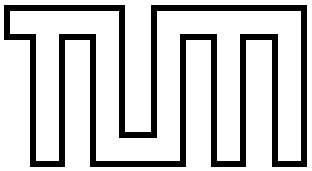


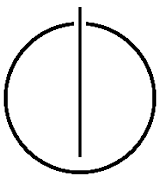
FAKULTÄT FÜR INFORMATIK

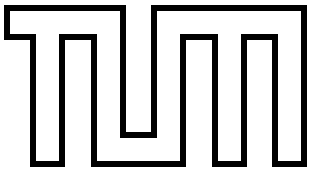
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Verified decision procedures for the equivalence of regular expressions

Maximilian Haslbeck





FAKULTÄT FÜR INFORMATIK

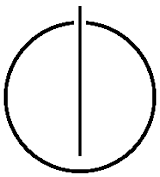
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Verified decision procedures for the equivalence of regular expressions

Verifizierte Entscheidungprozeduren für die Äquivalenz von Regulären Ausdrücken

Author:	Maximilian Haslbeck
Supervisor:	Prof. Dr. Tobias Nipkow
Advisor:	Prof. Dr. Tobias Nipkow
Date:	10. Juli, 2013



Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.
I assure the single handed composition of this bachelor's thesis only supported by declared resources.

München, den 10. Juli, 2013 Maximilian Haslbeck

Abstract

Many procedures for deciding regular expression equivalence were proposed. We examine four methods that do not rely on automata but work directly on regular expressions: based on derivatives, partial derivatives and two closely related versions of pointed regular expressions. We formalize and verify them in the ITP Isabelle. The methods can be formulated in a generic algorithm, for which we show soundness, termination and completeness.

Contents

Abstract **vii**

1 Introduction **1**

2 Foundations **3**
2.1 Languages and Regular Expressions 3
2.2 Derivatives of Regular Expressions 4
2.3 Partial Derivatives of Regular Expressions 5
2.4 Link between partial Derivatives and Derivatives 7

3 Pointed Regular Expressions **9**
3.1 Asperti 9
3.2 Fischer et al. 12
3.3 Synthesis 12

4 Bisimulation **15**
4.1 Generic bisimulation 15
4.1.1 Bisimulation 17
4.1.2 Computing the Bisimulation Closure 17
4.2 Applied bisimulation 20

5 Conclusion **21**

Bibliography **23**

1 Introduction

Equivalence of regular expression still is a ongoing topic in computer science. Despite the task being relatively old in the past few years various methods have been proposed and verified in different interactive theorem provers (ITP).

We examine four of these algorithms and condense them into a generic form. We formalize and verify this generic algorithm in the ITP Isabelle [6] and obtain four executable desicion procedures for regular expression equivalence. We build our development upon the work done by Krauss and Nipkow [5]

In Section 2 we revise the standard notions of regular languages, their left quotients and regular expressions. We revise the concept of derivatives of regular expressions due to Brzozowski [3] and partial derivatives of regular expressions due to Antimirov [1]. After that we obtain an upper bound for the number of partial derivatives and give a link to derivatives.

Pointed regular expressions (PRE) are discussed in Section 3: we first formalize and verify one kind of PRE along the lines of Asperti [2]. After that we introduce a version of PRE by Fischer et al. [4] and show that they are closely related.

In Section 3 we define bisimulations and show how to use them for testing regular expression equivalence. Then we come up with a generic algorithm in order to integrate the four introduced ways of handling regular expressions. We show soundness, termination and completeness of the algorithm and finally instantiate it by the four methods.

2 Foundations

2.1 Languages and Regular Expressions

Definition 2.1 *A language is a set of words; a word is a list of characters over an alphabet Σ. We define operations concatenation @@, power A^n and Kleene star * on languages:*
$A \mathbin{@@} B = \{xs \mathbin{@} ys \mid xs \in A \wedge ys \in B\}$
$A^0 = \{[]\}$
$A^{n+1} = A \mathbin{@@} A^n$
$A^* = (\bigcup_n A^n)$

Definition 2.2 *Also we define the left quotient of a language w.r.t. a single character a and lift it to a word w:*
Deriv x A $= \{xs \mid x{\cdot}xs \in A\}$
Derivs xs A $= \{ys \mid xs \mathbin{@} ys \in A\}$

It is easy to see that as an alternative to checking whether a word w is in a language we can check whether the empty word ([]) is in the left quotient w.r.t w.

Lemma 2.3 $(w \in A) = ([] \in \mathit{Derivs}\ w\ A)$

Definition 2.4 *Regular expressions are defined as a recursive datatype α rexp. we use the canonical definition with constructors $+$ for alternative, $\cdot$ for concatenation, * for Kleene star, $\emptyset$ for the additive and ε for the multiplicative identities.*

datatype *'a rexp* $= \emptyset \mid \varepsilon \mid$ *Atom 'a* $\mid$ *op* $+$ (*'a rexp*) (*'a rexp*)
$\mid$ *op* $\cdot$ (*'a rexp*) (*'a rexp*) $\mid$ *Star* (*'a rexp*)

Definition 2.5 *On this datatype we define a function atoms which returns the set of atoms in a regular expression and a function alph which determines the number of atoms in a regular expression. alph is also called the alphabetic width of a regular expression.*

$\mathit{atoms}\ \emptyset = \emptyset$	$\mathit{alph}(\emptyset) = 0$
$\mathit{atoms}\ \varepsilon = \emptyset$	$\mathit{alph}(\varepsilon) = 0$
$\mathit{atoms}\ (\mathit{Atom}\ a) = \{a\}$	$\mathit{alph}(\mathit{Atom}\ a) = 1$
$\mathit{atoms}\ (r + s) = \mathit{atoms}\ r \cup \mathit{atoms}\ s$	$\mathit{alph}(r1 + r2) = \mathit{alph}(r1) + \mathit{alph}(r2)$
$\mathit{atoms}\ (r \cdot s) = \mathit{atoms}\ r \cup \mathit{atoms}\ s$	$\mathit{alph}(r1 \cdot r2) = \mathit{alph}(r1) + \mathit{alph}(r2)$
$\mathit{atoms}\ r^* = \mathit{atoms}\ r$	$\mathit{alph}(r1^*) = \mathit{alph}(r1)$

Note that the alphabetic width is not equal to the cardinality of the set of atoms, as we possibly count identical atoms multiple times.

Definition 2.6 *We define the language of a regular expression in the standard way.*

$$\mathfrak{L}(\emptyset) = \emptyset \qquad \mathfrak{L}(r + s) = \mathfrak{L}(r) \cup \mathfrak{L}(s)$$
$$\mathfrak{L}(\varepsilon) = \{[]\} \qquad \mathfrak{L}(r \cdot s) = \mathfrak{L}(r) \mathbin{@@} \mathfrak{L}(s)$$
$$\mathfrak{L}(\mathit{Atom}\ a) = \{[a]\} \qquad \mathfrak{L}(r^*) = \mathfrak{L}(r)^*$$

Throughout the whole document we use $\mathfrak{L}(t)$ to denote the language of some term t regardless of which type t is, we leave the simple disambiguation task to the reader.
It is decidable whether the language of a regular expression contains the empty word. The function *nullable* determines this.

Definition 2.7 *(nullable)*

$$\mathit{nullable}\ \emptyset = \mathit{False} \qquad \mathit{nullable}\ (r1 + r2) = (\mathit{nullable}\ r1 \vee \mathit{nullable}\ r2)$$
$$\mathit{nullable}\ \varepsilon = \mathit{True} \qquad \mathit{nullable}\ (r1 \cdot r2) = (\mathit{nullable}\ r1 \wedge \mathit{nullable}\ r2)$$
$$\mathit{nullable}\ (\mathit{Atom}\ c) = \mathit{False} \qquad \mathit{nullable}\ r^* = \mathit{True}$$

By induction we obtain the characteristic property $\mathit{nullable}\ r = ([] \in \mathfrak{L}(r))$.
For convenience we say two languages are *conullable* if and only if both contain the empty word, as well as two regular expressions are *conullable* if and only if both are *nullable*.
Equality of two languages can be shown by coinduction. The following lemma is the key to the soundness proof of our decision procedures.

Lemma 2.8 *Let R be a binary relation between languages such that every pair in R is conullable and for every pair in R and symbol x the pair of its left quotients w.r.t. x is also in R. Then if (K, L) is in R the languages are the same.*
$[\![R\ K\ L;\ \bigwedge K\ L.\ R\ K\ L \Longrightarrow ([] \in K) = ([] \in L);\ \bigwedge K\ L\ x.\ R\ K\ L \Longrightarrow R\ (\mathit{Deriv}\ x\ K)\ (\mathit{Deriv}\ x\ L)]\!] \Longrightarrow K = L$

Example 2.9 *Consider the simple regular expression $(a + \varepsilon) \cdot (b^* \cdot a)$. Its associated language is {a,aa,ba,aba,bba,abba,...} and it is not nullable. Its atoms are {a,b} and its alphabetic width is 3.*

2.2 Derivatives of Regular Expressions

Derivatives of regular expressions were first introduced by Brzozowski [3] in 1964. A derivative *deriv a r* is a regular expression that describes a language equal to the left quotient of *r* w.r.t. a symbol *a*. The derivatives are implemented in the function *deriv*

Definition 2.10 *deriv c* $\emptyset = \emptyset$
deriv c $\varepsilon = \emptyset$
deriv c $(Atom\ c') = (\mathsf{if}\ c = c'\ \mathsf{then}\ \varepsilon\ \mathsf{else}\ \emptyset)$
deriv c $(r1 + r2) =$ *deriv c r1* $+$ *deriv c r2*
deriv c $(r1 \cdot r2) = (\mathsf{if}\ nullable\ r1\ \mathsf{then}\ deriv\ c\ r1 \cdot r2 + deriv\ c\ r2\ \mathsf{else}\ deriv\ c\ r1 \cdot r2)$
deriv c $r^* =$ *deriv c r* $\cdot\ r^*$

By induction on the form of r we obtain the characteristic property:

Lemma 2.11 $\mathfrak{L}(deriv\ c\ r) = Deriv\ c\ (\mathfrak{L}(r))$

So we can use derivatives of regular expressions to calculate the left quotient of a language. We can lift the derivative in natural form to words.
As we do not operate on the language itself, which might be infinite, but on the regular expression and a finite word we obtain a computable matcher:

Definition 2.12 *matcher r s* $=$ *nullable* $(derivs\ s\ r)$

Again by induction we obtain the property *matcher r s* $= (s \in \mathfrak{L}(r))$.

Example 2.13 *(continuing)*
The derivative of $(a + \varepsilon) \cdot (b^* \cdot a)$ *w.r.t symbol a is* $(\emptyset{+}\varepsilon){\cdot}(b^*{\cdot}a) + \emptyset{\cdot}b^*{\cdot}a + \varepsilon$.

Brzozowski showed, that the set of derivatives modulo ACI of the operator $+$ (i.e. associativety, commutativity and idempotence) is finite. (Theorem 5.2 in [3]) He uses this result to construct a DFA from a regular expression r.
Formalizing this idea is not part of this Bachelor's Thesis, but would be needed to form directly a decision procedure for the equivalence of regular expressions based on derivatives.

2.3 Partial Derivatives of Regular Expressions

As an addition to derivatives Antimirov [1] introduced the notion of partial derivatives as its "non-deterministic generalization".

Definition 2.14 *pderiv c* $\emptyset = \emptyset$
pderiv c $\varepsilon = \emptyset$
pderiv c $(Atom\ c') = (\mathsf{if}\ c = c'\ \mathsf{then}\ \{\varepsilon\}\ \mathsf{else}\ \emptyset)$
pderiv c $(r1 + r2) =$ *pderiv c r1* $\cup$ *pderiv c r2*
pderiv c $(r1 \cdot r2) = (\mathsf{if}\ nullable\ r1\ \mathsf{then}\ Timess\ (pderiv\ c\ r1)\ r2 \cup pderiv\ c\ r2\ \mathsf{else}\ Timess\ (pderiv\ c\ r1)\ r2)$
pderiv c $r^* = Timess\ (pderiv\ c\ r)\ r^*$

In the definition of the partial derivatives we use the helper funciton *Timess*, which lifts the concatenation of regular expressions to sets of regular expressions.

As before for derivatives, we obtain the left quotient of r using partial derivatives of r. Every element p in *pderiv* is one "part" of the left Quotient of r, hence the name.

Lemma 2.15 *Deriv c* $(\mathfrak{L}(r)) = \bigcup \{\mathfrak{L}(p) \mid p \in pderiv\ c\ r\}$

Here we can also see the tight connection between partial derivatives and derivatives — the union of languages of all elements of a partial derivative form the language of a derivative:

Lemma 2.16 $\bigcup \{\mathfrak{L}(p) \mid p \in pderiv\ c\ r\} = \mathfrak{L}(deriv\ c\ r)$

Operations like *atom*, *nullable*, $\mathfrak{L}(r)$ and *pderiv* can be lifted to sets of regular expressions:

Definition 2.17 *Atoms* $R \equiv \bigcup_{r \in R} atoms\ r$
Nullable $R \equiv \exists r \in R.\ nullable\ r$
$\mathfrak{L}(R) \equiv \bigcup_{r \in R} \mathfrak{L}(r)$
Pderiv a $R \equiv \bigcup_{r \in R} pderiv\ a\ r$

Regular expressions, derivatives and partial derivatives were already formalized in Isabelle in the way presented here by Krauss and Nipkow [5]. We base further development on their work.

Example 2.18 *(continuing)*
The partial derivatives of $(a+\varepsilon)\cdot(b^*\cdot a)$ *w.r.t symbol a are* $\{\varepsilon\cdot b^*\cdot a, \varepsilon\}$.

As one of the main theoretical results of Antimirov's paper [1], he shows that there are only finitely many partial derivatives for a given regular expression r and gives an upper bound for the cardinality of the set:

Theorem 2.19 (upperbound for partial derivatives) $|pderivs\text{-}lang\ UNIV1\ r| \leq alph(r)$
$|pderivs\text{-}lang\ UNIV\ r| \leq alph(r) + 1$

Here *UNIV* is the language containing all words and *UNIV1* the language with out empty word. *pderivs-lang A r* is the set of partial derivatives w.r.t. every word in A. The theorem states that there are no more partial derivatives of r then $alph(r)+1$.

To prove this theorem we first show the first goal, only considering non empty words and prove the upperbound $alph(r)$. By later adding the partial derivative of the empty word again we obtain desired result. The pending goal can be proved by induction on the structure of the regular expression r. The proof is along the lines of Theorem 3.4 in [1].

As a second theoretical result Antimirov shows that the partial derivative of a regular expression r only consists of a possibly empty and bounded concatenation of subexpressions of r (Theroem 3.8 in [1]). So given r one could calculate all subexpressions and every partial derivative can be represented in a quite condensed datastructure: a list of references to subterms of r. With that at hand, testing for equality of two partial derivatives can be implemented quite efficiently.

2.4 Link between partial Derivatives and Derivatives

As we have seen in Lemma 2.16 the language of a derivative can be composed by the language of the elements of the partial derivatives. More specifically we can give a function *set-of* that maps a derivative to its partial derivative:

Definition 2.20 $\mathit{set\text{-}of}\ (r1 + r2) = \mathit{set\text{-}of}\ r1 \cup \mathit{set\text{-}of}\ r2$
$\mathit{set\text{-}of}\ (r1 \cdot r2) = \mathit{Timess}\ (\mathit{set\text{-}of}\ r1)\ r2$
$\mathit{set\text{-}of}\ \emptyset = \emptyset$
$\mathit{set\text{-}of}\ \varepsilon = \{\varepsilon\}$
$\mathit{set\text{-}of}\ (\mathit{Atom}\ v) = \{\mathit{Atom}\ v\}$
$\mathit{set\text{-}of}\ v^* = \{v^*\}$

The desired property $\mathit{set\text{-}of}\ (\mathit{deriv}\ a\ r) = \mathit{pderiv}\ a\ r$ can be shown by induction.

Example 2.21 *(continuing)*
Applying set-of on the derivative obtained in Example 2.13 yields $\{\varepsilon \cdot b^* \cdot a, \varepsilon\}$*, which is (as expected) the set of partial derivatives from Example 2.18.*

As we have seen, to obtain a finite number of states when using derivatives we need to quotient the derivatives w.r.t. a suitable abstraction function. The function *set-of* can be seen as such an abstraction function. It maps a derivative of a regular expression to an equivalence class. As we know that the set of partial derivatives is finite and we already have obtained an upper bound in Theorem 2.19 for its cardinality, we now use this to show that the number of derivatives modulo *set-of* is finite and give an upper bound:

Lemma 2.22 $|\bigcup_w \{\mathit{set\text{-}of}\ (\mathit{derivs}\ w\ r)\}| \leq 1 + 2^{\mathit{alph}(r)}$

Note that we use the first part of Theorem 2.19 and then add the *set-of* the derivative w.r.t. the empty word (which is only one element).
Lemma 2.22 enables formulating an terminating algorithm for testing regular expression equivalence based on derivatives. Before doing this we take a detour to some alternative to derivatives.

3 Pointed Regular Expressions

3.1 Asperti

Asperti [2] formalizes an algorithm for testing regular expression equivalence, based on the notion of pointed regular expression.

According to Asperti derivatives have the weakness, that they are forced to quotient derivatives w.r.t a suitable notion of equivalence in order to get a finite number of states. Using pointed regular expressions one could avoid these problems. Asperti formalized the notion of pointed regular expressions in the ITP Matita. Here we reconstruct his work in Isabelle.

A *pointed item* (*pitem*) is a regular expression where every atom can be either pointed or not. Intuitively speaking, a point marks a position that is reached after reading some prefix of the input string. As we can not mark the end of the regular expression, a *pointed regular expression* (*pre*) is a *pitem i* with an additional flag *b*.

Definition 3.1 *We model the pitem as a regular expression over pairs of a bool and an element of type* α*. The pre in turn is defined as a pair* $\langle i,b \rangle$ *of a bool b and a pitem i.*

α *pitem* $= (bool \times \alpha)$ *rexp*

α *pre* $= (\alpha$ *pitem* $\times \alpha)$

We introduce the notion of the *carrier* $|i|$ of a *pitem i*, which is the regular expression obtained when deleting all the points of *i*. This definition can be lifted to *pre* also.

The language of a *pre* is the the language associated with its *pitem* and the empty word if its flag is true. The language of a *pitem* is the union of the language of all its points. In turn the language associated with a point is the language that is accepted by the regular expression starting from its position. This semantics is captured in the following definitions:

Definition 3.2 $\mathfrak{L}(\emptyset) = \emptyset$

$\mathfrak{L}(\varepsilon) = \emptyset$

$\mathfrak{L}(Atom\ (False, \_)) = \emptyset$

$\mathfrak{L}(Atom\ (True, a)) = \{[a]\}$

$\mathfrak{L}(r + s) = \mathfrak{L}(r) \cup \mathfrak{L}(s)$

$\mathfrak{L}(r \cdot s) = \mathfrak{L}(r)$ @@ $\mathfrak{L}(|s|) \cup \mathfrak{L}(s)$

$\mathfrak{L}(r^*) = \mathfrak{L}(r)$ @@ $\mathfrak{L}(|r|)^*$

Definition 3.3 $\mathfrak{L}((i, b)) = (\mathit{if}\ b\ \mathit{then}\ \mathfrak{L}(i) \cup \{[]\}\ \mathit{else}\ \mathfrak{L}(i))$

As a result we obtain the property $([] \in \mathfrak{L}((i, b))) = b$

So in Asperti's pointed regular expressions the points in the regular expressions are just before the atoms. Thus the points represent positions where we could start reading the next character. When we now actually read a new character a all those positions have to be considered. All pointed atoms with associated character a are then active and have to be broadcast. Unpointed atoms and atoms with associated character different from a just get ignored. If the character is the same as the pointed character, the point gets broadcast.
Asperti introduces the operation *eclose* ($\bullet$), that broadcasts a point into an *pitem*. As the point could reach the end of the pitem this operation generates a *pre*.

Definition 3.4 $\bullet\, \emptyset = (\emptyset, \mathit{False})$
$\bullet\, \varepsilon = (\varepsilon, \mathit{True})$
$\bullet\, \mathit{Atom}\ (\mathit{False}, x) = (\mathit{Atom}\ (\mathit{True}, x), \mathit{False})$
$\bullet\, \mathit{Atom}\ (\mathit{True}, x) = (\mathit{Atom}\ (\mathit{True}, x), \mathit{False})$
$\bullet\, i1 + i2 = \bullet\, i1 \oplus \bullet\, i2$
$\bullet\, i1 \cdot i2 = \bullet\, i1 \odot (i2, \mathit{False})$
$\bullet\, i^* = (\mathit{fst}\ (\bullet\, i)^*, \mathit{True})$

With the operators lifted from *pitem* to *pre*:

Definition 3.5 $r1 \oplus r2 = (\mathit{fst}\ r1 + \mathit{fst}\ r2, \mathit{snd}\ r1 \vee \mathit{snd}\ r2)$
$r1 \odot r2 = (\mathsf{let}\ (i', b') = \mathsf{if}\ \mathit{snd}\ r1\ \mathsf{then}\ \mathit{con\text{-}item\text{-}pre}\ (\mathit{fst}\ r1)\ (\bullet\, \mathit{fst}\ r2)\ \mathsf{else}\ (\mathit{fst}\ r1 \cdot \mathit{fst}\ r2, \mathit{False})\ \mathsf{in}\ (i', b' \vee \mathit{snd}\ r2))$
$r^\star = (\mathsf{let}\ (i, b) = r\ \mathsf{in}\ \mathsf{if}\ b\ \mathsf{then}\ (\mathit{fst}\ (\bullet\, i)^*, \mathit{True})\ \mathsf{else}\ (i^*, \mathit{False}))$

We can characterize the semantics of *eclose* and the lifted operators. Furthermore by induction we can show, that the carrier does not change after applying eclose.

Theorem 3.6 $\mathfrak{L}(\bullet\, i) = \mathfrak{L}(i) \cup \mathfrak{L}(|i|)$
$\mathfrak{L}(e1 \oplus e2) = \mathfrak{L}(e1) \cup \mathfrak{L}(e2)$
$\mathfrak{L}(p1 \odot p2) = \mathfrak{L}(p1)\ @@\ \mathfrak{L}(|\mathit{fst}\ p2|) \cup \mathfrak{L}(p2)$
$\mathfrak{L}(p^\star) = \mathfrak{L}(p)\ @@\ \mathfrak{L}(|\mathit{fst}\ p|)^*$
$|\mathit{fst}\ (\bullet\, i)| = |i|$

The operation *eclose* can be lifted to a *pre* $\langle i,b\rangle$ (*preclose*). The effect is, that a point is broadcast into the *pitem* i and the resulting flag is true if a point reaches the end of the item or the flag b already was true.

Example 3.7 *(continuing, cf. example 2 of [2])*
When we shift a point into our example regular expression $(a{+}\varepsilon){\cdot}(b^*{\cdot}a)$ *we start working in parallel on the first occurence of a (where the point stops), and on* ε *that gets traversed. We have hence reached the end of* $a{+}\varepsilon$ *and must pursue broadcasting inside* $(b^*{\cdot}a)$*. Thus, the point is allowed to both enter the star, and to traverse it, stopping in front of a. No point reached the end of* $(b^*{\cdot}a)$ *hence, no further propagation is possible. In conclusion:*
$\bullet(a{+}\varepsilon){\cdot}(b^*{\cdot}a) = \langle(\bullet a{+}\varepsilon){\cdot}(\bullet b^*{\cdot}\bullet a), \mathit{false}\rangle$.

Then we define a move operation, that takes a *pitem* and a character and returns a new *pre*. The semantics of this operation is that we obtain the *pre* after reading one character.

Definition 3.8 $move\ c\ \emptyset = (\emptyset, False)$
$move\ c\ \varepsilon = (\varepsilon, False)$
$move\ c\ (Atom\ (False, x)) = (Atom\ (False, x), False)$
$move\ c\ (Atom\ (True, x)) = (Atom\ (False, x), c = x)$
$move\ c\ (i1 + i2) = move\ c\ i1 \oplus move\ c\ i2$
$move\ c\ (i1 \cdot i2) = move\ c\ i1 \odot move\ c\ i2$
$move\ c\ i^* = move\ c\ i^\star$

What happens: for every point in front of an atom a, let the point traverse the character and broadcast it. All points preceding a character different from a must be removed.

Example 3.9 *(continuing, cf. example 3 of [2])*
When we now consider the pitem we obtained in the example before: $(\bullet a{+}\varepsilon){\cdot}(\bullet b^*{\cdot}\bullet a)$. *Let us apply the move operation w.r.t. the character a. For a we have two possible positions (the other point will be erased). The first point gets broadcasted inside* $(b^*{\cdot}a)$ *like in the previous step and stops before the b and the a. The second point reaches the end of the term. In total:*
$move\ a\ (\bullet a{+}\varepsilon){\cdot}(\bullet b^*{\cdot}\bullet a) = \langle (a{+}\varepsilon){\cdot}(\bullet b^*{\cdot}\bullet a), true\rangle$.

With the knowledge of Theorem 3.6 it is easy to show that the *move* operation corresponds to the left quotient. We obtain the characteristic property of *move*:

Theorem 3.10 $\mathfrak{L}(move\ a\ i) = Deriv\ a\ (\mathfrak{L}(i))$

Moreover we can show that the *carrier* stays unchanged after a *move* operation:

Lemma 3.11 $|move\ a\ i| = |i|$

When we consider all *pitem i* with carrier *r*, every atom can be pointed or not. So the set of all those *i* is finite and has cardinality $2^{alph(r)}$.

Lemma 3.12 (Upperbound for pitem) $|\{s \mid |s| = r\}| = 2^{alph(r)}$

Lifting this to pre just doubles the upper bound (as there are two possible values for the flag). With Lemma 3.12 we obtain an upper bound for the number of states in the DFA of r.
This is essentially all we need: a computable variant of the left quotient (by Lemma 3.10) and an upper bound for the number of states (by Lemma 3.12).

3.2 Fischer et al.

In *"A Play on Regular Expressions"* Fischer et al. [4] give an elegant way of working directly with regular expressions in a functional programming setting. They present a matching algorithms on regular expression with boolean marks indicating where in the regular expression the matching process has arrived. In contrast to the version of pointed regular expressions we have just examined, those symbols are marked that have just been read. One can easily suspect that the two versions are closely related.

We transfer the implementation of Fischer et al., which is basically only 15 lines of code. We extend the setup to the additive identity $\emptyset$ and model the "postpointed" regular expressions just as the "prepointed" regular expressions: by a regular expression over a pair of a bool and an element of a type α. (Just the interpretation is different). As in turn "postpointed" regular expressions can't be pointed in the beginning we complete the setup by forming a pre as pair of a pitem and a flag. As this flag is only true before the first read of a character (points can not reach the beginning again) the flag was left away by Fischer et al.

The two operations *final* and *shift* are what we take from Fischer et. al.

Definition 3.13 $\mathit{final}\ \varepsilon = \mathit{False}$
$\mathit{final}\ \emptyset = \mathit{False}$
$\mathit{final}\ (\mathit{Atom}\ (b, \_)) = b$
$\mathit{final}\ (p + q) = (\mathit{final}\ p \vee \mathit{final}\ q)$
$\mathit{final}\ (p \cdot q) = (\mathit{final}\ p \wedge \mathit{nullable}\ q \vee \mathit{final}\ q)$
$\mathit{final}\ r^* = \mathit{final}\ r$

Definition 3.14 $\mathit{shift}\ \_\ \varepsilon\ \_ = \varepsilon$
$\mathit{shift}\ m\ (\mathit{Atom}\ (\_, x))\ c = \mathit{Atom}\ (m \wedge x = c, x)$
$\mathit{shift}\ m\ (p + q)\ c = \mathit{shift}\ m\ p\ c + \mathit{shift}\ m\ q\ c$
$\mathit{shift}\ m\ (p \cdot q)\ c = \mathit{shift}\ m\ p\ c \cdot \mathit{shift}\ (m \wedge \mathit{nullable}\ p \vee \mathit{final}\ p)\ q\ c$
$\mathit{shift}\ m\ r^*\ c = \mathit{shift}\ (m \vee \mathit{final}\ r)\ r\ c^*$

Proving facts about the "postpointed" regular expressions requieres defining a semantics of those, i.e. what the associated language is. As the atoms are marked on the back they have to be broadcast throughout the regular expression to bring them in front of the next atom to be read. Then we can determine the language associated with the term. This broadcasting sounds quite familiar to the broadcasting we did for the "prepointed" regular expressions. To save some work and to show that the two constructs are very closely related we generalize them and then use the results for the first to show correctness of the second.

3.3 Synthesis

The *move* operation on the "prepointed" regular expressions can be carried out in two steps: First the points that preceed an atom with the character we are currently reading

traverse the atom. This step takes a "prepointed" and returns a "postpointed" regular expression.
Secondly, the points that succeed an atom get broadcast so that they again stand in front of some atom. This operation translates a "postpointed" into a "prepointed" regular expression.

Example 3.15 *(continuing)*
Consider again the "prepointed" regular expression $(\bullet a{+}\varepsilon){\cdot}(\bullet b^*{\cdot}\bullet a)$. *After executing operation traverse w.r.t. character a we obtain* $(a\bullet{+}\varepsilon){\cdot}(b^*{\cdot}a\bullet)$.
The operation broadcast afterwards pushes the first point into $(b^*{\cdot}a)$ *and the second point reaches the end of the term, thus yields:* $\langle(a{+}\varepsilon){\cdot}(\bullet b^*{\cdot}\bullet a), \mathit{true}\rangle$.

The following definitions implements the idea:

Definition 3.16 (traverse)
$\mathit{traverse}\ \emptyset\ c = \emptyset$
$\mathit{traverse}\ \varepsilon\ c = \varepsilon$
$\mathit{traverse}\ (\mathit{Atom}\ (\mathit{False}, x))\ c = \mathit{Atom}\ (\mathit{False}, x)$
$\mathit{traverse}\ (\mathit{Atom}\ (\mathit{True}, x))\ c = \mathit{Atom}\ (x = c, x)$
$\mathit{traverse}\ (p + q)\ c = \mathit{traverse}\ p\ c + \mathit{traverse}\ q\ c$
$\mathit{traverse}\ (p \cdot q)\ c = \mathit{traverse}\ p\ c \cdot \mathit{traverse}\ q\ c$
$\mathit{traverse}\ r^*\ c = \mathit{traverse}\ r\ c^*$

Definition 3.17 (broadcast)
$\mathit{broadcast}\ \varepsilon\ m = (\varepsilon, m)$
$\mathit{broadcast}\ \emptyset\ m = (\emptyset, \mathit{False})$
$\mathit{broadcast}\ (\mathit{Atom}\ (b, x))\ \mathit{True} = (\mathit{Atom}\ (\mathit{True}, x), b)$
$\mathit{broadcast}\ (\mathit{Atom}\ (b, x))\ \mathit{False} = (\mathit{Atom}\ (\mathit{False}, x), b)$
$\mathit{broadcast}\ (p + q)\ m = (\mathit{fst}\ (\mathit{broadcast}\ p\ m) + \mathit{fst}\ (\mathit{broadcast}\ q\ m), \mathit{snd}\ (\mathit{broadcast}\ p\ m) \vee \mathit{snd}\ (\mathit{broadcast}\ q\ m))$
$\mathit{broadcast}\ (p \cdot q)\ m = (\mathsf{let}\ (i1, b1) = \mathit{broadcast}\ p\ m; (i2, y) = \mathit{broadcast}\ q\ b1\ \mathsf{in}\ (i1 \cdot i2, y))$
$\mathit{broadcast}\ r^*\ m = (\mathsf{let}\ (i1, b1) = \mathit{broadcast}\ r\ m\ \mathsf{in\ if}\ b1\ \mathsf{then}\ (\mathit{fst}\ (\mathit{broadcast}\ r\ \mathit{True})^*, \mathit{True})\ \mathsf{else}\ (i1^*, m \wedge \neg\ \mathit{nullable}\ r \vee \mathit{False}))$

We can show that *move* and *shift* indeed are compositions of *traverse* and *broadcast*:

Lemma 3.18 $\mathit{move}\ c\ r = \mathit{broadcast}\ (\mathit{traverse}\ r\ c)\ \mathit{False}$
$\mathit{shift}\ m\ r\ c = \mathit{traverse}\ (\mathit{fst}\ (\mathit{broadcast}\ r\ m))\ c$

Now we define the semantics of a "postpointed" *pre* by first performing a broadcast on its *pitem*, obtaining a *pre* and then using the already defined semantics for *pre*. In the case that the flag is set to true we add the language of the *carrier* of the *pitem* to the language; this corresponds to a point preceeding the regular expression.

Definition 3.19 $\mathfrak{L}(l) = \mathfrak{L}(\mathit{broadcast}\ (\mathit{fst}\ l)\ \mathit{False}) \cup (\mathsf{if}\ \mathit{snd}\ l\ \mathsf{then}\ \mathfrak{L}(\lfloor \mathit{fst}\ l \rfloor)\ \mathsf{else}\ \emptyset)$

With this definition we can use Lemma 3.10 to show an equivalent property for Fischer et al.

Lemma 3.20 $\mathfrak{L}((\mathit{shift}\ b\ i\ a, \mathit{False})) = \mathit{Deriv}\ a\ \mathfrak{L}((i, b))$

Analogously to the "prepointed" regular expressions (Lemma 3.12) we obtain an upper bound for the number of possible states in a DFA when using "postpointed" regular expressions.

4 Bisimulation

Krauss and Nipkow [5] describe an algorithm for testing equivalence of regular expressions r and s based on derivatives: it incrementally constructs the relation of all (*derivs w r*, *derivs w s*). Provided that r and s are equivalent, all pairs of states have to behave the same w.r.t acceptance of input words. If the procedure terminates, it either yields a bisimulation relation (then r and s are equivalent) or it must find a pair (r', s') where one of them is a final state and the other one not.

They proceed by defining a normalization function for regular expressions and then state the algorithm explicitely. Furthermore they show soundness for the procedure.

We generalize this idea for an decision procedure for regular expression equivalence and not only allow to use derivatives but any transition function $\delta\ a\ r$. This function can be lifted from characters a to words ($\delta\ w\ r$). Then we incrementally construct the relation of all ($\delta\ w\ r$, $\delta\ w\ s$). Provided that r and s are equivalent all pairs of states must be equivalent w.r.t accepting the word w. If this procedure termintates, it either returns a bisimulation relation or a counterexample for equivalence. Additionally we show that the procedure actually terminates.

4.1 Generic bisimulation

In the sections before we saw four different transition functions δ and arguments why the number of states of a DFA induced by it is finite. We now generalize the equivalence checker *check-eqv* by Krauss and Nipkow [5].

For partial derivatives and the versions of pointed regular expression the development is pretty straightforward. But for derivatives we have to add an abstraction: when enumerating all pairs of states we should not visit two pairs such that they are in the same equivalence class w.r.t to an abstraction function. Otherwise we loose the finiteness of the state space.

In Isabelle we can use *locales* to define local scopes. We define the locale *g-bisim* which requiers certain operations with some properties. Inside the scope of the locale we then can refer to these operations and properties. From these we can derive new properties and methods. Later this locale can be instantiated by providing the requiered operations satisfying the requiered properties.

Here is a full listing of the requiered operations with their types.

Definition 4.1 *lang* :: $'b \Rightarrow 'a\ list\ set$

prep :: $'a\ rexp \Rightarrow 'b$

δ :: $'a \Rightarrow 'b \Rightarrow 'b$

sink :: $'b \Rightarrow 'b$

nullo :: *'b* ⇒ *bool*
atoms :: *'b* ⇒ *'a set*
reachable :: *'a rexp* ⇒ *'a rexp* ⇒ (*'c* × *'c*) *set*
ub-card-reachable :: *'a rexp* ⇒ *'a rexp* ⇒ *nat*
abstr :: *'b* ⇒ *'c*

Those operations have to satisfy the following properties.

Definition 4.2 *step-sink*: *"δ a (sink r) = sink r"*
sink-conullable-aux: *"nullo (sink r) = nullo (sink s)"*
atoms-sink: *"atoms (sink r) ⊆ atoms r"*
nullable-iff-pointed: *"nullo r = ([] ∈ 𝔏(r))"*
lang-pointed: *"𝔏(δ a r) = Deriv a 𝔏(r)"*
step-no-occurrence: *"x ∉ atoms r ⟹ 𝔏(δ x r) = 𝔏(sink r)"*
atoms: *"atoms r1 = atoms (prep r1)"*
atoms-step: *"atoms (δ a r) ⊆ atoms r"*
lang-prep: *"𝔏(r1) = 𝔏(prep r1)"*
lang-abstr: *"abstr r = abstr s ⟹ 𝔏(r) = 𝔏(s)"*
sink-sink: *"sink i = sink (sink i)"*
sink-step: *"sink (δ a r) = sink r"*
reachable-finite: *"finite (reachable r1 s1)"*
reachable-bounded: *"|reachable r1 s1| ≤ ub-card-reachable r1 s1"*
inset: *"(abstr (prep r1), abstr (prep s1)) ∈ reachable r1 s1"*
staysinset: *"(abstr h, abstr i) ∈ reachable r1 s1 ⟹ (abstr (δ a h), abstr (δ a i)) ∈ reachable r1 s1"*

In the following we describe the most important operations and properties that are needed:
The specific algorithms perform not necessarily on regular expressions but on terms of some type β. Thus we require a operation $\mathfrak{L}$ to convert a term of β into a language. In the beginning of the equivalence computation the regular expressions are converted to terms of β. This is done by a *prep* operation, that must generate a term with the same associated language (assured by property *lang-prep*).
Furthermore there must be a δ function that performs an analogous operation on β as the left quotient (property *lang-pointed*). For the case that a symbol can not be read by a term t of β we provide a *sink* state, to which the δ operation directs in that case. An operation *nullo* on β has to determine whether the empty word is in the corresponding language.
For the termination argument we require an abstraction function *abstr*, that only merges terms of β with same corresponding language (property *lang-abstr*). One must specify a superset *reachable* of the abstracted state space and a bound *ub-card-reachable* for its cardinality. To ensure termination it must hold, that *reachable* contains the starting pair and is closed under the δ operation.

4.1.1 Bisimulation

The predicate *is-bisimulation* checks for a given alphabet Σ whether a given list *ps* of pairs of states is a bisimulation.

We demand that for every pair (r, s) in *ps* and every character *a* there is already an element $(\delta\ a\ r, \delta\ a\ s)$ in *ps* modulo some abstraction function. Furthermore the predicate requieres that all expressions in *ps* contain only atoms from Σ, that for every pair (r, s) in *ps* either both contain the empty word or none and that for every two pairs in *ps* the *sink* of the first components are the same, and the *sink* of the second components are the same.

Definition 4.3 *is-g-bisimulation as ps* =
$(\forall (r, s) \in set\ ps.$
$\quad (\forall (r', s') \in set\ ps.$
$\quad\quad sink\ r = sink\ r' \wedge sink\ s = sink\ s') \wedge$
$\quad atoms\ r \cup atoms\ s \subseteq set\ as \wedge$
$\quad conullable\ r\ s \wedge$
$\quad (\forall a \in set\ as.\ (\delta\ a\ r, \delta\ a\ s) \in_a set\ ps))$

So with Lemma 2.8 we can show that if we have a list *ps* of pairs of regular expressions that forms a bisimulation containing the pair (rS, sS) then languages of rS and sS are equal.

Lemma 4.4 $[\![is\text{-}g\text{-}bisimulation\ as\ ps; (rS, sS) \in set\ ps]\!] \Longrightarrow \mathfrak{L}(rS) = \mathfrak{L}(sS)$

4.1.2 Computing the Bisimulation Closure

In order to get the method computable, we have to define a computable bisimulation closure. What we do to show that two regular expressions *r* , *s* are equal is:

We start from the pair (r, s) and successively generate all pairs $(\delta\ a\ r', \delta\ a\ s')$. This is done with a worklist based algorithm: it uses a worklist *ws* and a list of pairs that are already done *ps*. In every step we take the first pair (r', s') out of *ws* and put it into *ps*. For this pair (r', s') we compute for every character *a* from Σ (where Σ is the set of atoms in *r* and *s*) $\delta\ a\ r'$ and $\delta\ a\ s'$. Every $(\delta\ a\ r', \delta\ a\ s')$ computed like this is added to *ws*. Such a pair is only inserted if it is not yet inside *ws* or *ps*. "Is inside" (denoted by $\in_a$) means, that there is already an element inside the set that is equal w.r.t. the abstraction function.

These ideas are implemented in the *bistep* function:

Definition 4.5 *bistep as* (ws, ps) =
$(\mathsf{let}\ (r, s) = hd\ ws; ps' = (r, s) \cdot ps;$
$\quad succs = map\ (\lambda a.\ (\delta\ a\ r, \delta\ a\ s))\ as;$
$\quad new = [p \leftarrow succs\ .\ \neg\ p \in_a (set\ ps \cup set\ ws)]$
$\mathsf{in}\ (remdups\text{-}a\ new\ @\ tl\ ws, ps'))$

To the intermediate list *new* only elements are added that to not have an equal element in *ws* or *ps*. *remdups-a* removes duplicates w.r.t. *abstr* from that list. We keep on iterating this step until the worklist is empty or we find a counter example.

Definition 4.6 *test* $(ws, ps) = (\mathsf{case}\ ws\ \mathsf{of}\ [] \Rightarrow \mathit{False} \mid (p, q)\cdot xs \Rightarrow \mathit{nullo}\ p = \mathit{nullo}\ q)$

Definition 4.7 *closure as* $=$ *while-option test* (*bistep as*)

To show that this closure computation is sound we need to specify an invariant for the execution of *bistep*. Given some starting pair (r, s) and alphabet Σ a pair of a worklist *ws* and donelist *ps* satisfy the invariant *pre-bisim* if several conditions are fulfilled:

the pair (r, s) is already done or in the worklist. every pair already worked off or in the work list has the same *sink*. every pair worked off is conullable, i.e. either both elements of the pair contain the empty word or none. And for every pair worked off its successors are in the worklist or also worked off.

Definition 4.8 *pre-bisim as r s* $=$
$(\lambda(ws, ps).$
$(r, s) \in \mathit{set}\ ws \cup \mathit{set}\ ps \wedge$
$(\forall (r', s') \in \mathit{set}\ ws \cup \mathit{set}\ ps.$
$\mathit{sink}\ r = \mathit{sink}\ r' \wedge \mathit{sink}\ s = \mathit{sink}\ s') \wedge$
$(\forall (r, s) \in \mathit{set}\ ws \cup \mathit{set}\ ps.\ \mathit{atoms}\ r \cup \mathit{atoms}\ s \subseteq \mathit{set}\ as) \wedge$
$(\forall (r, s) \in \mathit{set}\ ps.$
$\mathit{nullo}\ r = \mathit{nullo}\ s \wedge$
$(\forall a \in \mathit{set}\ as.\ (\delta\ a\ r, \delta\ a\ s) \in_a (\mathit{set}\ ps \cup \mathit{set}\ ws))))$

We show that this invariant holds after the execution of a *bistep* if it also held before and the test is true:

Lemma 4.9 $[\![\mathit{pre\text{-}bisim\ as\ r\ s\ st}; \mathit{test\ st}]\!] \Longrightarrow \mathit{pre\text{-}bisim\ as\ r\ s}\ (\mathit{bistep\ as\ st})$

It is easy to see that *pre-bisim* holds initially and we have already shown that it is an invariant of bistep. Assuming that the process terminates with an empty worklist, we can derive that the invariant also holds after termination. The invariant holding with an empty worklist implies that we obtained a bisimulation. Finally with Lemma 4.4 we have that *r* and *s* are equal.

Theorem 4.10 (closure sound)
$[\![\mathit{closure\ as}\ ([(r, s)], []) = \mathit{Some}\ ([], ps); \mathit{atoms}\ r \cup \mathit{atoms}\ s \subseteq \mathit{set}\ as]\!] \Longrightarrow \mathfrak{L}(r) = \mathfrak{L}(s)$

Having proven soundness we can now proceed with showint termination and completeness.

The set of pairs $(\delta\ w\ r, \delta\ w\ s)$ is finite, or better the cardinality is bounded. So we can prove termination for *closure*. If we can show that in every step of the closure computation one new pair is added to the donelist, we have an argument for termination. This can be ensured by another invariant for bistep *bistep-invariant*: it makes sure that the worklist and the donelist contain only distinct elements w.r.t. abstraction, that worklist and donelist are disjoint w.r.t abstraction and that the abstraction of any pair considered is in the superset *reachable*.

Definition 4.11 *bistep-invariant r1′ r2′ =*
(λ*s*. *distinct-a* (*fst s*) ∧
distinct-a (*snd s*) ∧
disj-resp-same (*set* (*fst s*)) (*set* (*snd s*)) ∧
(∀ (*r′*, *s′*)∈*set* (*fst s*) ∪ *set* (*snd s*).
(*abstr r′*, *abstr s′*) ∈ *reachable r1′ r2′*))

The ranking function for the closure computation is defined as:

Definition 4.12 *closure-rank r s x = ub-card-reachable r s − |snd x|*

The invariant *bistep-invariant* holds and the ranking function decreases if the invariant held before the execution and the test is true:

Lemma 4.13 ⟦*bistep-invariant r1′ r2′ s*; *test s*⟧ ⟹ *bistep-invariant r1′ r2′* (*bistep as s*)
⟦*bistep-invariant r1′ r2′ s*; *test s*⟧ ⟹ *closure-rank r1′ r2′* (*bistep as s*) < *closure-rank r1′ r2′ s*

With the ranking function decreasing after every iteration the closure computation terminates.

Lemma 4.14 (closure terminates)
⟦*prep r1′ = r1*; *prep r2′ = r2*⟧ ⟹ ∃ *P*. *closure as* ([(*r1*, *r2*)], []) = *Some P*

We further can conclude that the closure computation is complete:

Theorem 4.15 (closure complete)
⟦𝔏(*r1′*) = 𝔏(*r2′*); *prep r1′ = r1*; *prep r2′ = r2*⟧ ⟹ ∃ *ps*. *closure as* ([(*r1*, *r2*)], []) = *Some* ([], *ps*)

Finally we integrate closure computation into a function *check-eqv* that takes two regular expressions and returns whether they are equal. For this it converts the input to a equivalent term on which the respective method computes, determines a list of symbols of *r* and *s* (by a function *add-atoms*) and checks whether *closure* has terminated with an empty worklist.

Definition 4.16 *check-eqv r s =*
(*case closure* (*add-atom r* (*add-atom s* [])) ([(*prep r*, *prep s*)], []) *of*
None ⇒ *False* | *Some* ([], *x*) ⇒ *True* | *Some* (*a*·*list*, *x*) ⇒ *False*)

As a final result we can show soundness and completeness for the overall procedure as a corollary of Theorem 4.10 and Theorem 4.15:

Corollary 4.17 (𝔏(*r*) = 𝔏(*s*)) = *check-eqv r s*

4.2 Applied bisimulation

With the generic version of bisimulation in place we can instantiate it by various methods.

For a equivalence checker based on derivatives we use the standard operations on regular expressions, *deriv* as the δ operation and *set-of* as an abstraction function. The superset and the upper bound for its cardinality follow from Lemma 2.22: As we consider pairs of sets we have to multiply the cardinalities and obtain an upper bound $((1::'d) + (2::'d)^{alph(r)}) * ((1::'d) + (2::'d)^{alph(s)})$. The main property needed is Lemma 2.11.

We lift operations on regular expressions to sets of regular expressions (as in Definition 2.17) and do not use any abstraction (*abstr* is identity) to obtain an equivalence checker based on partial derivatives. The states that operations are performed on is a pair of subsets of the set of partial derivatives. Resulting from Lemma 2.19 and the subset construction we obtain an upper bound $2^{1 + alph(r)} * 2^{1 + alph(s)}$ for this set. That a step can be performed is ensured by Lemma 2.15.

Those two methods essentially do the same, as working on derivatives modulo *set-of* is the same as working on subsets of the set of partial derivatives. Nevertheless the development enables using derivatives when a suitable abstraction function is provided. Suitable in that way, that finiteness of the state space is proven also. For example one could use the normalization *norm* on regular expressions by Krauss and Nipkow [5] when provining a proof for the finiteness of the set of derivatives modulo *norm*. Two different upper bounds for the cardinality of *reachable* were obtained. Further investigation could reveal whether the better upper bound can be proven also for the method with pderivs or the method with derivs really has some advantage.

For pointed regular expressions we lift the appropriate operations to pre and use no abstraction function. The finiteness of the state space was already discussed and successful steps are ensured by Lemma 3.10 respectively Lemma 3.20. For an upper bound of the cardinality of *reachable* we obtain $2^{2 + alph(a) + alph(b)}$ which is the same as for partial derivatives.

As for the "postpointed" regular expressions, the flag is only *true* before reading the first character, one could obtain a smaller set *reachable* and thus a better upper bound for its cardinality (similar to the one for derivatives).

Following from Corollary 4.17 proof obligations $\mathfrak{L}(r) = \mathfrak{L}(s)$ may now be reduced to any variant of *check-eqv r s*. By executing the equivalence checker the proof is reduced to a computation.

5 Conclusion

We examined four different ways of determining the left quotient of a regular expression. We proved an upper bound for number of partial derivatives of a regular expression, and thus obtained also an upper bound for the number of derivatives modulo an abstraction function *set-of*.

We reconstructed the formalization of pointed regular expressions from [2]. Afterwards we showed that this can be used to verify another version of pointed regular expressions, which is closely related.

Finally we generalized the equivalence checker by Krauss and Nipkow [5] and applied it for our four methods. By defining the algorithm appropriately it was possible to not only show soundness but also termination and completeness.

Bibliography

[1] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291 – 319, 1996.

[2] A. Asperti. A compact proof of decidability for regular expression equivalence. In L. Beringer and A. Felty, editors, *Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 283–298. Springer Berlin Heidelberg, 2012.

[3] J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11:481–494, 1964.

[4] S. Fischer, F. Huch, and T. Wilke. A play on regular expressions: functional pearl. In P. Hudak and S. Weirich, editors, *ICFP*, pages 357–368. ACM, 2010.

[5] A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Automated Reasoning*, 49:95–106, 2012. published online March 2011.

[6] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.