

# Regular Sets, Expressions, Derivatives and Relation Algebra

Alexander Krauss, Tobias Nipkow,  
Chunhan Wu, Xingyuan Zhang and Christian Urban

May 14, 2024

## Abstract

This is a library of constructions on regular expressions and languages. It provides the operations of concatenation, Kleene star and left-quotients of languages. A theory of derivatives and partial derivatives is provided. Arden's lemma and finiteness of partial derivatives is established. A simple regular expression matcher based on Brozowski's derivatives is proved to be correct. An executable equivalence checker for regular expressions is verified; it does not need automata but works directly on regular expressions. By mapping regular expressions to binary relations, an automatic and complete proof method for (in)equalities of binary relations over union, concatenation and (reflexive) transitive closure is obtained.

For an exposition of the equivalence checker for regular and relation algebraic expressions see the paper by Krauss and Nipkow [3].

Extended regular expressions with complement and intersection are also defined and an equivalence checker is provided.

## Contents

<b>1</b>	<b>Regular sets</b>	<b>2</b>
1.1	$(@@)$ . . . . .	3
1.2	$A^n$ . . . . .	4
1.3	<i>star</i> . . . . .	5
1.4	Left-Quotients of languages . . . . .	7
1.5	Shuffle product . . . . .	9
1.6	Arden's Lemma . . . . .	10
<b>2</b>	<b>Regular expressions</b>	<b>12</b>
2.1	Term ordering . . . . .	14
<b>3</b>	<b>Normalizing Derivative</b>	<b>16</b>
3.1	Normalizing operations . . . . .	16

<b>4</b>	<b>Deciding Regular Expression Equivalence</b>	<b>18</b>
4.1	Bisimulation between languages and regular expressions . . .	18
4.2	Closure computation . . . . .	19
4.3	Bisimulation-free proof of closure computation . . . . .	20
4.4	The overall procedure . . . . .	21
<b>5</b>	<b>Regular Expressions as Homogeneous Binary Relations</b>	<b>22</b>
<b>6</b>	<b>Proving Relation (In)equalities via Regular Expressions</b>	<b>23</b>
<b>7</b>	<b>Basic constructions on regular expressions</b>	<b>25</b>
7.1	Reverse language . . . . .	25
7.2	Substituting characters in a language . . . . .	25
7.3	Subword language . . . . .	28
7.4	Fragment language . . . . .	29
7.5	Various regular expression constructions . . . . .	30
<b>8</b>	<b>Derivatives of regular expressions</b>	<b>32</b>
8.1	Brzozowski's derivatives of regular expressions . . . . .	33
8.2	Antimirov's partial derivatives . . . . .	33
8.3	Relating left-quotients and partial derivatives . . . . .	35
8.4	Relating derivatives and partial derivatives . . . . .	35
8.5	Finiteness property of partial derivatives . . . . .	35
<b>9</b>	<b>Deciding Regular Expression Equivalence (2)</b>	<b>40</b>
9.1	Closure computation . . . . .	42
9.2	The overall procedure . . . . .	44
9.3	Termination and Completeness . . . . .	45
<b>10</b>	<b>Extended Regular Expressions</b>	<b>47</b>
<b>11</b>	<b>Deciding Equivalence of Extended Regular Expressions</b>	<b>48</b>
11.1	Term ordering . . . . .	48
11.2	Normalizing operations . . . . .	49
11.3	Derivative . . . . .	50
11.4	Bisimulation between languages and regular expressions . . .	50
11.5	Closure computation . . . . .	52
11.6	The overall procedure . . . . .	53

## 1 Regular sets

```
theory Regular-Set
imports Main
begin
```

**type-synonym** 'a lang = 'a list set

**definition** conc :: 'a lang ⇒ 'a lang ⇒ 'a lang (**infixr** @@ 75) **where**  
A @@ B = {xs@ys | xs ys. xs:A & ys:B}

checks the code preprocessor for set comprehensions

**export-code** conc **checking** SML

**overloading** lang-pow == compow :: nat ⇒ 'a lang ⇒ 'a lang

**begin**

**primrec** lang-pow :: nat ⇒ 'a lang ⇒ 'a lang **where**

lang-pow 0 A = {[]}

lang-pow (Suc n) A = A @@ (lang-pow n A)

**end**

for code generation

**definition** lang-pow :: nat ⇒ 'a lang ⇒ 'a lang **where**

lang-pow-code-def [code-abbrev]: lang-pow = compow

**lemma** [code]:

lang-pow (Suc n) A = A @@ (lang-pow n A)

lang-pow 0 A = {[]}

**by** (simp-all add: lang-pow-code-def)

**hide-const** (open) lang-pow

**definition** star :: 'a lang ⇒ 'a lang **where**

star A = (⋃ n. A  $\overset{\sim}{\sim}$  n)

### 1.1 (@@)

**lemma** concI[simp,intro]: u : A ⇒ v : B ⇒ u@v : A @@ B

**by** (auto simp add: conc-def)

**lemma** concE[elim]:

**assumes** w ∈ A @@ B

**obtains** u v **where** u ∈ A v ∈ B w = u@v

**using** assms **by** (auto simp: conc-def)

**lemma** conc-mono: A ⊆ C ⇒ B ⊆ D ⇒ A @@ B ⊆ C @@ D

**by** (auto simp: conc-def)

**lemma** conc-empty[simp]: **shows** {} @@ A = {} **and** A @@ {} = {}

**by** auto

**lemma** conc-epsilon[simp]: **shows** [[]] @@ A = A **and** A @@ [[]] = A

**by** (simp-all add:conc-def)

**lemma** conc-assoc: (A @@ B) @@ C = A @@ (B @@ C)

**by** (auto elim!: concE) (simp only: append-assoc[symmetric] concI)

**lemma** *conc-Un-distrib*:

**shows**  $A @@ (B \cup C) = A @@ B \cup A @@ C$

**and**  $(A \cup B) @@ C = A @@ C \cup B @@ C$

**by** *auto*

**lemma** *conc-UNION-distrib*:

**shows**  $A @@ \bigcup (M \text{ ' } I) = \bigcup ((\%i. A @@ M i) \text{ ' } I)$

**and**  $\bigcup (M \text{ ' } I) @@ A = \bigcup ((\%i. M i @@ A) \text{ ' } I)$

**by** *auto*

**lemma** *conc-subset-lists*:  $A \subseteq \text{lists } S \implies B \subseteq \text{lists } S \implies A @@ B \subseteq \text{lists } S$

**by** (*fastforce simp: conc-def in-lists-conv-set*)

**lemma** *Nil-in-conc[simp]*:  $\square \in A @@ B \longleftrightarrow \square \in A \wedge \square \in B$

**by** (*metis append-is-Nil-conv concE concI*)

**lemma** *concI-if-Nil1*:  $\square \in A \implies xs : B \implies xs \in A @@ B$

**by** (*metis append-Nil concI*)

**lemma** *conc-Diff-if-Nil1*:  $\square \in A \implies A @@ B = (A - \{\square\}) @@ B \cup B$

**by** (*fastforce elim: concI-if-Nil1*)

**lemma** *concI-if-Nil2*:  $\square \in B \implies xs : A \implies xs \in A @@ B$

**by** (*metis append-Nil2 concI*)

**lemma** *conc-Diff-if-Nil2*:  $\square \in B \implies A @@ B = A @@ (B - \{\square\}) \cup A$

**by** (*fastforce elim: concI-if-Nil2*)

**lemma** *singleton-in-conc*:

$[x] : A @@ B \longleftrightarrow [x] : A \wedge \square : B \vee \square : A \wedge [x] : B$

**by** (*fastforce simp: Cons-eq-append-conv append-eq-Cons-conv conc-Diff-if-Nil1 conc-Diff-if-Nil2*)

## 1.2 $A^n$

**lemma** *lang-pow-add*:  $A \text{ } \overset{\sim}{\sim} (n + m) = A \text{ } \overset{\sim}{\sim} n @@ A \text{ } \overset{\sim}{\sim} m$

**by** (*induct n*) (*auto simp: conc-assoc*)

**lemma** *lang-pow-empty*:  $\{\} \text{ } \overset{\sim}{\sim} n = (\text{if } n = 0 \text{ then } \{\square\} \text{ else } \{\})$

**by** (*induct n*) *auto*

**lemma** *lang-pow-empty-Suc[simp]*:  $(\{\} :: 'a \text{ lang}) \text{ } \overset{\sim}{\sim} \text{Suc } n = \{\}$

**by** (*simp add: lang-pow-empty*)

**lemma** *conc-pow-comm*:

**shows**  $A @@ (A \text{ } \overset{\sim}{\sim} n) = (A \text{ } \overset{\sim}{\sim} n) @@ A$

**by** (*induct n*) (*simp-all add: conc-assoc[symmetric]*)

**lemma** *length-lang-pow-ub*:  
 $\forall w \in A. \text{length } w \leq k \implies w : A^{\sim n} \implies \text{length } w \leq k*n$   
**by**(*induct n arbitrary: w*) (*fastforce simp: conc-def*)+

**lemma** *length-lang-pow-lb*:  
 $\forall w \in A. \text{length } w \geq k \implies w : A^{\sim n} \implies \text{length } w \geq k*n$   
**by**(*induct n arbitrary: w*) (*fastforce simp: conc-def*)+

**lemma** *lang-pow-subset-lists*:  $A \subseteq \text{lists } S \implies A^{\sim n} \subseteq \text{lists } S$   
**by**(*induct n*)(*auto simp: conc-subset-lists*)

**lemma** *empty-pow-add*:  
**assumes**  $\square \in A \ s \in A^{\sim n}$   
**shows**  $s \in A^{\sim (n + m)}$   
**using** *assms*  
**apply**(*induct m arbitrary: n*)  
**apply**(*auto simp add: concI-if-Nil1*)  
**done**

### 1.3 *star*

**lemma** *star-subset-lists*:  $A \subseteq \text{lists } S \implies \text{star } A \subseteq \text{lists } S$   
**unfolding** *star-def* **by**(*blast dest: lang-pow-subset-lists*)

**lemma** *star-if-lang-pow[simp]*:  $w : A^{\sim n} \implies w : \text{star } A$   
**by** (*auto simp: star-def*)

**lemma** *Nil-in-star[iff]*:  $\square : \text{star } A$   
**proof** (*rule star-if-lang-pow*)  
**show**  $\square : A^{\sim 0}$  **by** *simp*  
**qed**

**lemma** *star-if-lang[simp]*: **assumes**  $w : A$  **shows**  $w : \text{star } A$   
**proof** (*rule star-if-lang-pow*)  
**show**  $w : A^{\sim 1}$  **using**  $\langle w : A \rangle$  **by** *simp*  
**qed**

**lemma** *append-in-starI[simp]*:  
**assumes**  $u : \text{star } A$  **and**  $v : \text{star } A$  **shows**  $u@v : \text{star } A$   
**proof** –  
**from**  $\langle u : \text{star } A \rangle$  **obtain**  $m$  **where**  $u : A^{\sim m}$  **by** (*auto simp: star-def*)  
**moreover**  
**from**  $\langle v : \text{star } A \rangle$  **obtain**  $n$  **where**  $v : A^{\sim n}$  **by** (*auto simp: star-def*)  
**ultimately have**  $u@v : A^{\sim (m+n)}$  **by** (*simp add: lang-pow-add*)  
**thus** *?thesis* **by** *simp*  
**qed**

**lemma** *conc-star-star*:  $\text{star } A @@ \text{star } A = \text{star } A$   
**by** (*auto simp: conc-def*)

```

lemma conc-star-comm:
  shows  $A @@ star A = star A @@ A$ 
unfolding star-def conc-pow-comm conc-UNION-distrib
by simp

lemma star-induct[consumes 1, case-names Nil append, induct set: star]:
assumes  $w : star A$ 
  and  $P []$ 
  and step:  $!!u v. u : A \implies v : star A \implies P v \implies P (u@v)$ 
shows  $P w$ 
proof -
  { fix  $n$  have  $w : A \overset{\sim}{\sim} n \implies P w$ 
    by (induct n arbitrary: w) (auto intro: <P []> step star-if-lang-pow) }
  with  $\langle w : star A \rangle$  show  $P w$  by (auto simp: star-def)
qed

lemma star-empty[simp]:  $star \{\} = \{\{\}$ 
by (auto elim: star-induct)

lemma star-epsilon[simp]:  $star \{\{\} = \{\{\}$ 
by (auto elim: star-induct)

lemma star-idemp[simp]:  $star (star A) = star A$ 
by (auto elim: star-induct)

lemma star-unfold-left:  $star A = A @@ star A \cup \{\{\}$  (is  $?L = ?R$ )
proof
  show  $?L \subseteq ?R$  by (rule, erule star-induct) auto
qed auto

lemma concat-in-star:  $set ws \subseteq A \implies concat ws : star A$ 
by (induct ws) simp-all

lemma in-star-iff-concat:
   $w \in star A = (\exists ws. set ws \subseteq A \wedge w = concat ws)$ 
  (is  $- = (\exists ws. ?R w ws)$ )
proof
  assume  $w : star A$  thus  $\exists ws. ?R w ws$ 
  proof induct
    case Nil have  $?R [] []$  by simp
    thus ?case ..
  next
    case (append u v)
    then obtain  $ws$  where  $set ws \subseteq A \wedge v = concat ws$  by blast
    with append have  $?R (u@v) (u\#ws)$  by auto
    thus ?case ..
  qed
next

```

**assume**  $\exists us. ?R w us$  **thus**  $w : star A$   
**by** (*auto simp: concat-in-star*)  
**qed**

**lemma** *star-conv-concat*:  $star A = \{concat\ ws | ws. set\ ws \subseteq A\}$   
**by** (*fastforce simp: in-star-iff-concat*)

**lemma** *star-insert-eps*[*simp*]:  $star (insert\ []\ A) = star(A)$   
**proof** –

**{ fix** *us*  
**have**  $set\ us \subseteq insert\ []\ A \implies \exists vs. concat\ us = concat\ vs \wedge set\ vs \subseteq A$   
**(is**  $?P \implies \exists vs. ?Q\ vs)$   
**proof**  
**let**  $?vs = filter\ (\%u. u \neq [])\ us$   
**show**  $?P \implies ?Q\ ?vs$  **by** (*induct us*) *auto*  
**qed**  
**} thus** *?thesis* **by** (*auto simp: star-conv-concat*)  
**qed**

**lemma** *star-unfold-left-Nil*:  $star A = (A - \{\epsilon\}) @@ (star A) \cup \{\epsilon\}$   
**by** (*metis insert-Diff-single star-insert-eps star-unfold-left*)

**lemma** *star-Diff-Nil-fold*:  $(A - \{\epsilon\}) @@ star A = star A - \{\epsilon\}$   
**proof** –

**have**  $\epsilon \notin (A - \{\epsilon\}) @@ star A$  **by** *simp*  
**thus** *?thesis* **using** *star-unfold-left-Nil* **by** *blast*  
**qed**

**lemma** *star-decom*:

**assumes**  $a: x \in star A\ x \neq []$   
**shows**  $\exists a\ b. x = a @ b \wedge a \neq [] \wedge a \in A \wedge b \in star A$   
**using** *a* **by** (*induct rule: star-induct*) (*blast*)+

**lemma** *star-pow*:

**assumes**  $s \in star A$   
**shows**  $\exists n. s \in A \overset{\sim}{\sim} n$   
**using** *assms*  
**apply** (*induct*)  
**apply** (*rule-tac x=0 in exI*)  
**apply** (*auto*)  
**apply** (*rule-tac x=Suc n in exI*)  
**apply** (*auto*)  
**done**

## 1.4 Left-Quotients of languages

**definition** *Deriv* ::  $'a \Rightarrow 'a\ lang \Rightarrow 'a\ lang$   
**where**  $Deriv\ x\ A = \{xs. x\#xs \in A\}$

**definition**  $Derivs :: 'a list \Rightarrow 'a lang \Rightarrow 'a lang$   
**where**  $Derivs\ xs\ A = \{ ys.\ xs\ @\ ys \in A \}$

**abbreviation**

$Derivss :: 'a list \Rightarrow 'a lang\ set \Rightarrow 'a lang$   
**where**  
 $Derivss\ s\ As \equiv \bigcup (Derivs\ s\ 'As)$

**lemma**  $Deriv-empty[simp]$ :  $Deriv\ a\ \{\} = \{\}$   
**and**  $Deriv-epsilon[simp]$ :  $Deriv\ a\ \{\}\{\} = \{\}$   
**and**  $Deriv-char[simp]$ :  $Deriv\ a\ \{[b]\} = (if\ a = b\ then\ \{\}\{\} else\ \{\})$   
**and**  $Deriv-union[simp]$ :  $Deriv\ a\ (A \cup B) = Deriv\ a\ A \cup Deriv\ a\ B$   
**and**  $Deriv-inter[simp]$ :  $Deriv\ a\ (A \cap B) = Deriv\ a\ A \cap Deriv\ a\ B$   
**and**  $Deriv-compl[simp]$ :  $Deriv\ a\ (-A) = -\ Deriv\ a\ A$   
**and**  $Deriv-Union[simp]$ :  $Deriv\ a\ (Union\ M) = Union(Deriv\ a\ 'M)$   
**and**  $Deriv-UN[simp]$ :  $Deriv\ a\ (UN\ x:I.\ S\ x) = (UN\ x:I.\ Deriv\ a\ (S\ x))$   
**by** (*auto simp: Deriv-def*)

**lemma**  $Der-conc [simp]$ :  
**shows**  $Deriv\ c\ (A\ @@\ B) = (Deriv\ c\ A)\ @@\ B \cup (if\ [] \in A\ then\ Deriv\ c\ B\ else\ \{\})$   
**unfolding**  $Deriv-def\ conc-def$   
**by** (*auto simp add: Cons-eq-append-conv*)

**lemma**  $Deriv-star [simp]$ :  
**shows**  $Deriv\ c\ (star\ A) = (Deriv\ c\ A)\ @@\ star\ A$   
**proof** –  
**have**  $Deriv\ c\ (star\ A) = Deriv\ c\ (\{\}\{\} \cup A\ @@\ star\ A)$   
**by** (*metis star-unfold-left sup commute*)  
**also have**  $\dots = Deriv\ c\ (A\ @@\ star\ A)$   
**unfolding**  $Deriv-union$  **by** (*simp*)  
**also have**  $\dots = (Deriv\ c\ A)\ @@\ (star\ A) \cup (if\ [] \in A\ then\ Deriv\ c\ (star\ A)\ else\ \{\})$   
**by** *simp*  
**also have**  $\dots = (Deriv\ c\ A)\ @@\ star\ A$   
**unfolding**  $conc-def\ Deriv-def$   
**using**  $star-decom$  **by** (*force simp add: Cons-eq-append-conv*)  
**finally show**  $Deriv\ c\ (star\ A) = (Deriv\ c\ A)\ @@\ star\ A .$   
**qed**

**lemma**  $Deriv-diff[simp]$ :  
**shows**  $Deriv\ c\ (A - B) = Deriv\ c\ A - Deriv\ c\ B$   
**by**(*auto simp add: Deriv-def*)

**lemma**  $Deriv-lists[simp]$ :  $c : S \Longrightarrow Deriv\ c\ (lists\ S) = lists\ S$   
**by**(*auto simp add: Deriv-def*)

**lemma**  $Derivs-simps [simp]$ :



**shows**  $Derivs [] A = A$   
**and**  $Derivs (c \# s) A = Derivs s (Deriv c A)$   
**and**  $Derivs (s1 @ s2) A = Derivs s2 (Derivs s1 A)$   
**unfolding**  $Derivs-def$   $Deriv-def$  **by** *auto*

**lemma** *in-fold-Deriv*:  $v \in fold\ Deriv\ w\ L \longleftrightarrow w @ v \in L$   
**by** (*induct w arbitrary: L*) (*simp-all add: Deriv-def*)

**lemma** *Derivs-alt-def* [*code*]:  $Derivs\ w\ L = fold\ Deriv\ w\ L$   
**by** (*induct w arbitrary: L*) *simp-all*

**lemma** *Deriv-code* [*code*]:  
 $Deriv\ x\ A = tl\ 'Set.filter\ (\lambda xs.\ case\ xs\ of\ x' \# \_ \Rightarrow x = x' \mid \_ \Rightarrow False)\ A$   
**by** (*auto simp: Deriv-def Set.filter-def image-iff tl-def split: list.splits*)

## 1.5 Shuffle product

**definition** *Shuffle* (**infixr**  $\parallel$  80) **where**  
 $Shuffle\ A\ B = \bigcup \{shuffles\ xs\ ys \mid xs\ ys.\ xs \in A \wedge ys \in B\}$

**lemma** *Deriv-Shuffle*[*simp*]:  
 $Deriv\ a\ (A \parallel B) = Deriv\ a\ A \parallel B \cup A \parallel Deriv\ a\ B$   
**unfolding** *Shuffle-def* *Deriv-def* **by** (*fastforce simp: Cons-in-shuffles-iff neq-Nil-conv*)

**lemma** *shuffle-subset-lists*:  
**assumes**  $A \subseteq lists\ S$   $B \subseteq lists\ S$   
**shows**  $A \parallel B \subseteq lists\ S$   
**unfolding** *Shuffle-def* **proof** *safe*  
**fix**  $x$  **and**  $zs\ xs\ ys :: 'a\ list$   
**assume**  $zs: zs \in shuffles\ xs\ ys$   $x \in set\ zs$  **and**  $xs \in A$   $ys \in B$   
**with** *assms* **have**  $xs \in lists\ S$   $ys \in lists\ S$  **by** *auto*  
**with**  $zs$  **show**  $x \in S$  **by** (*induct xs ys arbitrary: zs rule: shuffles.induct*) *auto*  
**qed**

**lemma** *Nil-in-Shuffle*[*simp*]:  $[] \in A \parallel B \longleftrightarrow [] \in A \wedge [] \in B$   
**unfolding** *Shuffle-def* **by** *force*

**lemma** *shuffle-Un-distrib*:  
**shows**  $A \parallel (B \cup C) = A \parallel B \cup A \parallel C$   
**and**  $A \parallel (B \cup C) = A \parallel B \cup A \parallel C$   
**unfolding** *Shuffle-def* **by** *fast+*

**lemma** *shuffle-UNION-distrib*:  
**shows**  $A \parallel \bigcup (M\ 'I) = \bigcup ((\%i.\ A \parallel M\ i)\ 'I)$   
**and**  $\bigcup (M\ 'I) \parallel A = \bigcup ((\%i.\ M\ i \parallel A)\ 'I)$   
**unfolding** *Shuffle-def* **by** *fast+*

**lemma** *Shuffle-empty*[*simp*]:  
 $A \parallel \{\} = \{\}$

$\{\} \parallel B = \{\}$   
**unfolding** *Shuffle-def* **by** *auto*

**lemma** *Shuffle-eps[simp]*:

$A \parallel \{\} = A$

$\{\} \parallel B = B$

**unfolding** *Shuffle-def* **by** *auto*

## 1.6 Arden's Lemma

**lemma** *arden-helper*:

**assumes** *eq*:  $X = A @@@ X \cup B$

**shows**  $X = (A \rightsquigarrow \text{Suc } n) @@@ X \cup (\bigcup_{m \leq n}. (A \rightsquigarrow m) @@@ B)$

**proof** (*induct n*)

**case** *0*

**show**  $X = (A \rightsquigarrow \text{Suc } 0) @@@ X \cup (\bigcup_{m \leq 0}. (A \rightsquigarrow m) @@@ B)$

**using** *eq* **by** *simp*

**next**

**case** (*Suc n*)

**have** *ih*:  $X = (A \rightsquigarrow \text{Suc } n) @@@ X \cup (\bigcup_{m \leq n}. (A \rightsquigarrow m) @@@ B)$  **by** *fact*

**also have**  $\dots = (A \rightsquigarrow \text{Suc } n) @@@ (A @@@ X \cup B) \cup (\bigcup_{m \leq n}. (A \rightsquigarrow m) @@@ B)$

**using** *eq* **by** *simp*

**also have**  $\dots = (A \rightsquigarrow \text{Suc } (\text{Suc } n)) @@@ X \cup ((A \rightsquigarrow \text{Suc } n) @@@ B) \cup (\bigcup_{m \leq n}. (A \rightsquigarrow m) @@@ B)$

**by** (*simp add: conc-Un-distrib conc-assoc[symmetric] conc-pow-comm*)

**also have**  $\dots = (A \rightsquigarrow \text{Suc } (\text{Suc } n)) @@@ X \cup (\bigcup_{m \leq \text{Suc } n}. (A \rightsquigarrow m) @@@ B)$

**by** (*auto simp add: atMost-Suc*)

**finally show**  $X = (A \rightsquigarrow \text{Suc } (\text{Suc } n)) @@@ X \cup (\bigcup_{m \leq \text{Suc } n}. (A \rightsquigarrow m) @@@ B)$

.

**qed**

**lemma** *Arden*:

**assumes**  $\square \notin A$

**shows**  $X = A @@@ X \cup B \longleftrightarrow X = \text{star } A @@@ B$

**proof**

**assume** *eq*:  $X = A @@@ X \cup B$

**{ fix** *w* **assume**  $w : X$

**let**  $?n = \text{size } w$

**from**  $\langle \square \notin A \rangle$  **have**  $\forall u \in A. \text{length } u \geq 1$

**by** (*metis Suc-eq-plus1 add-leD2 le-0-eq length-0-conv not-less-eq-eq*)

**hence**  $\forall u \in A \rightsquigarrow (?n+1). \text{length } u \geq ?n+1$

**by** (*metis length-lang-pow-lb nat-mult-1*)

**hence**  $\forall u \in A \rightsquigarrow (?n+1) @@@ X. \text{length } u \geq ?n+1$

**by** (*auto simp only: conc-def length-append*)

**hence**  $w \notin A \rightsquigarrow (?n+1) @@@ X$  **by** *auto*

**hence**  $w : \text{star } A @@@ B$  **using**  $\langle w : X \rangle$  **using** *arden-helper[OF eq, where n=?n]*

**by** (*auto simp add: star-def conc-UNION-distrib*)

**} moreover**

```

{ fix w assume w : star A @@ B
  hence  $\exists n. w \in A \overset{\sim}{\sim} n @@ B$  by (auto simp: conc-def star-def)
  hence w : X using arden-helper[OF eq] by blast
} ultimately show X = star A @@ B by blast
next
assume eq: X = star A @@ B
have star A = A @@ star A  $\cup$  {[]}
  by (rule star-unfold-left)
then have star A @@ B = (A @@ star A  $\cup$  {[]}) @@ B
  by metis
also have ... = (A @@ star A) @@ B  $\cup$  B
  unfolding conc-Un-distrib by simp
also have ... = A @@ (star A @@ B)  $\cup$  B
  by (simp only: conc-assoc)
finally show X = A @@ X  $\cup$  B
  using eq by blast
qed

```

lemma reversed-arden-helper:

```

assumes eq: X = X @@ A  $\cup$  B
shows X = X @@ (A  $\overset{\sim}{\sim}$  Suc n)  $\cup$  ( $\bigcup_{m \leq n}. B @@ (A \overset{\sim}{\sim} m)$ )
proof (induct n)
case 0
show X = X @@ (A  $\overset{\sim}{\sim}$  Suc 0)  $\cup$  ( $\bigcup_{m \leq 0}. B @@ (A \overset{\sim}{\sim} m)$ )
  using eq by simp
next
case (Suc n)
have ih: X = X @@ (A  $\overset{\sim}{\sim}$  Suc n)  $\cup$  ( $\bigcup_{m \leq n}. B @@ (A \overset{\sim}{\sim} m)$ ) by fact
also have ... = (X @@ A  $\cup$  B) @@ (A  $\overset{\sim}{\sim}$  Suc n)  $\cup$  ( $\bigcup_{m \leq n}. B @@ (A \overset{\sim}{\sim} m)$ )
using eq by simp
also have ... = X @@ (A  $\overset{\sim}{\sim}$  Suc (Suc n))  $\cup$  (B @@ (A  $\overset{\sim}{\sim}$  Suc n))  $\cup$  ( $\bigcup_{m \leq n}. B @@ (A \overset{\sim}{\sim} m)$ )
  by (simp add: conc-Un-distrib conc-assoc)
also have ... = X @@ (A  $\overset{\sim}{\sim}$  Suc (Suc n))  $\cup$  ( $\bigcup_{m \leq \text{Suc } n}. B @@ (A \overset{\sim}{\sim} m)$ )
  by (auto simp add: atMost-Suc)
finally show X = X @@ (A  $\overset{\sim}{\sim}$  Suc (Suc n))  $\cup$  ( $\bigcup_{m \leq \text{Suc } n}. B @@ (A \overset{\sim}{\sim} m)$ )
.
qed

```

theorem reversed-Arden:

```

assumes nemp: []  $\notin$  A
shows X = X @@ A  $\cup$  B  $\longleftrightarrow$  X = B @@ star A
proof
assume eq: X = X @@ A  $\cup$  B
{ fix w assume w : X
  let ?n = size w
  from <[]  $\notin$  A> have  $\forall u \in A. \text{length } u \geq 1$ 
    by (metis Suc-eq-plus1 add-leD2 le-0-eq length-0-conv not-less-eq-eq)

```

```

hence  $\forall u \in A^{\sim} (?n+1). \text{length } u \geq ?n+1$ 
  by (metis length-lang-pow-lb nat-mult-1)
hence  $\forall u \in X @ @ A^{\sim} (?n+1). \text{length } u \geq ?n+1$ 
  by (auto simp only: conc-def length-append)
hence  $w \notin X @ @ A^{\sim} (?n+1)$  by auto
hence  $w : B @ @ \text{star } A$  using  $\langle w : X \rangle$  using reversed-arden-helper[OF eq,
where  $n=?n]$ 
  by (auto simp add: star-def conc-UNION-distrib)
} moreover
{ fix  $w$  assume  $w : B @ @ \text{star } A$ 
  hence  $\exists n. w \in B @ @ A^{\sim} n$  by (auto simp: conc-def star-def)
  hence  $w : X$  using reversed-arden-helper[OF eq] by blast
} ultimately show  $X = B @ @ \text{star } A$  by blast
next
assume eq:  $X = B @ @ \text{star } A$ 
have  $\text{star } A = \{\}\cup \text{star } A @ @ A$ 
  unfolding conc-star-comm[symmetric]
  by (metis Un-commute star-unfold-left)
then have  $B @ @ \text{star } A = B @ @ (\{\}\cup \text{star } A @ @ A)$ 
  by metis
also have  $\dots = B \cup B @ @ (\text{star } A @ @ A)$ 
  unfolding conc-Un-distrib by simp
also have  $\dots = B \cup (B @ @ \text{star } A) @ @ A$ 
  by (simp only: conc-assoc)
finally show  $X = X @ @ A \cup B$ 
  using eq by blast
qed

end

```

## 2 Regular expressions

```

theory Regular-Exp
imports Regular-Set
begin

```

```

datatype (atoms: 'a) rexp =
  is-Zero: Zero |
  is-One: One |
  Atom 'a |
  Plus ('a rexp) ('a rexp) |
  Times ('a rexp) ('a rexp) |
  Star ('a rexp)

```

```

primrec lang :: 'a rexp => 'a lang where
lang Zero =  $\{\}$  |
lang One =  $\{\}\cup \{\}$  |
lang (Atom a) =  $\{[a]\}$  |
lang (Plus r s) = (lang r) Un (lang s) |

```

$lang (Times\ r\ s) = conc (lang\ r) (lang\ s) \mid$   
 $lang (Star\ r) = star(lang\ r)$

**abbreviation**  $(input)$  *regular-lang where regular-lang*  $A \equiv (\exists r. lang\ r = A)$

**primrec** *nullable* :: 'a *rexp*  $\Rightarrow$  *bool* **where**  
 $nullable\ Zero = False \mid$   
 $nullable\ One = True \mid$   
 $nullable (Atom\ c) = False \mid$   
 $nullable (Plus\ r1\ r2) = (nullable\ r1 \vee nullable\ r2) \mid$   
 $nullable (Times\ r1\ r2) = (nullable\ r1 \wedge nullable\ r2) \mid$   
 $nullable (Star\ r) = True$

**lemma** *nullable-iff* [*code-abbrev*]:  $nullable\ r \longleftrightarrow [] \in lang\ r$   
**by** (*induct r*) (*auto simp add: conc-def split: if-splits*)

**primrec** *rexp-empty* **where**  
 $rexp-empty\ Zero \longleftrightarrow True$   
 $\mid rexp-empty\ One \longleftrightarrow False$   
 $\mid rexp-empty (Atom\ a) \longleftrightarrow False$   
 $\mid rexp-empty (Plus\ r\ s) \longleftrightarrow rexp-empty\ r \wedge rexp-empty\ s$   
 $\mid rexp-empty (Times\ r\ s) \longleftrightarrow rexp-empty\ r \vee rexp-empty\ s$   
 $\mid rexp-empty (Star\ r) \longleftrightarrow False$

**lemma** *rexp-empty-iff* [*code-abbrev*]:  $rexp-empty\ r \longleftrightarrow lang\ r = \{\}$   
**by** (*induction r*) *auto*

Composition on rhs usually complicates matters:

**lemma** *map-map-rexp*:  
 $map-rexp\ f (map-rexp\ g\ r) = map-rexp (\lambda r. f (g\ r))\ r$   
**unfolding** *rexp.map-comp o-def ..*

**lemma** *map-rexp-ident*[*simp*]:  $map-rexp (\lambda x. x) = (\lambda r. r)$   
**unfolding** *id-def[symmetric] fun-eq-iff rexp.map-id id-apply* **by** (*intro allI refl*)

**lemma** *atoms-lang*:  $w : lang\ r \implies set\ w \subseteq atoms\ r$   
**proof**(*induction r arbitrary: w*)  
**case** *Times* **thus** ?*case* **by** *fastforce*  
**next**  
**case** *Star* **thus** ?*case* **by** (*fastforce simp add: star-conv-concat*)  
**qed** *auto*

**lemma** *lang-eq-ext*:  $(lang\ r = lang\ s) =$   
 $(\forall w \in lists(atoms\ r \cup atoms\ s). w \in lang\ r \longleftrightarrow w \in lang\ s)$   
**by** (*auto simp: atoms-lang[unfolded subset-iff]*)

**lemma** *lang-eq-ext-Nil-fold-Deriv*:  
**fixes**  $r\ s$

**defines**  $\mathfrak{B} \equiv \{(fold\ Deriv\ w\ (lang\ r),\ fold\ Deriv\ w\ (lang\ s)) \mid w.\ w \in lists\ (atoms\ r \cup atoms\ s)\}$   
**shows**  $lang\ r = lang\ s \iff (\forall (K, L) \in \mathfrak{B}.\ [] \in K \iff [] \in L)$   
**unfolding**  $lang\text{-}eq\text{-}ext\ \mathfrak{B}\text{-}def$  **by** (*subst (1 2) in-fold-Deriv[of [], simplified, symmetric] auto*)

## 2.1 Term ordering

**instantiation**  $rexp :: (order) \{order\}$   
**begin**

**fun**  $le\text{-}rexp :: ('a::order) rexp \Rightarrow ('a::order) rexp \Rightarrow bool$   
**where**

$le\text{-}rexp\ Zero\ - = True$   
 $| le\text{-}rexp\ -\ Zero = False$   
 $| le\text{-}rexp\ One\ - = True$   
 $| le\text{-}rexp\ -\ One = False$   
 $| le\text{-}rexp\ (Atom\ a)\ (Atom\ b) = (a \leq b)$   
 $| le\text{-}rexp\ (Atom\ -)\ - = True$   
 $| le\text{-}rexp\ -\ (Atom\ -) = False$   
 $| le\text{-}rexp\ (Star\ r)\ (Star\ s) = le\text{-}rexp\ r\ s$   
 $| le\text{-}rexp\ (Star\ -)\ - = True$   
 $| le\text{-}rexp\ -\ (Star\ -) = False$   
 $| le\text{-}rexp\ (Plus\ r\ r')\ (Plus\ s\ s') =$   
 $\quad (if\ r = s\ then\ le\text{-}rexp\ r'\ s'\ else\ le\text{-}rexp\ r\ s)$   
 $| le\text{-}rexp\ (Plus\ -)\ - = True$   
 $| le\text{-}rexp\ -\ (Plus\ -) = False$   
 $| le\text{-}rexp\ (Times\ r\ r')\ (Times\ s\ s') =$   
 $\quad (if\ r = s\ then\ le\text{-}rexp\ r'\ s'\ else\ le\text{-}rexp\ r\ s)$

**definition**  $less\text{-}eq\text{-}rexp$  **where**  $r \leq s \equiv le\text{-}rexp\ r\ s$

**definition**  $less\text{-}rexp$  **where**  $r < s \equiv le\text{-}rexp\ r\ s \wedge r \neq s$

**lemma**  $le\text{-}rexp\text{-}Zero$ :  $le\text{-}rexp\ r\ Zero \implies r = Zero$   
**by** (*induction r*) *auto*

**lemma**  $le\text{-}rexp\text{-}refl$ :  $le\text{-}rexp\ r\ r$   
**by** (*induction r*) *auto*

**lemma**  $le\text{-}rexp\text{-}antisym$ :  $\llbracket le\text{-}rexp\ r\ s; le\text{-}rexp\ s\ r \rrbracket \implies r = s$   
**by** (*induction r s rule: le-rexp.induct*) (*auto dest: le-rexp-Zero*)

**lemma**  $le\text{-}rexp\text{-}trans$ :  $\llbracket le\text{-}rexp\ r\ s; le\text{-}rexp\ s\ t \rrbracket \implies le\text{-}rexp\ r\ t$

**proof** (*induction r s arbitrary: t rule: le-rexp.induct*)

**fix**  $v\ t$  **assume**  $le\text{-}rexp\ (Atom\ v)\ t$  **thus**  $le\text{-}rexp\ One\ t$  **by** (*cases t*) *auto*  
**next**

**fix**  $s1\ s2\ t$  **assume**  $le\text{-}rexp\ (Plus\ s1\ s2)\ t$  **thus**  $le\text{-}rexp\ One\ t$  **by** (*cases t*) *auto*

```

next
  fix s1 s2 t assume le-rexp (Times s1 s2) t thus le-rexp One t by (cases t) auto
next
  fix s t assume le-rexp (Star s) t thus le-rexp One t by (cases t) auto
next
  fix v u t assume le-rexp (Atom v) (Atom u) le-rexp (Atom u) t
  thus le-rexp (Atom v) t by (cases t) auto
next
  fix v s1 s2 t assume le-rexp (Plus s1 s2) t thus le-rexp (Atom v) t by (cases t)
  auto
next
  fix v s1 s2 t assume le-rexp (Times s1 s2) t thus le-rexp (Atom v) t by (cases
  t) auto
next
  fix v s t assume le-rexp (Star s) t thus le-rexp (Atom v) t by (cases t) auto
next
  fix r s t
  assume IH:  $\bigwedge t. le-rexp r s \implies le-rexp s t \implies le-rexp r t$ 
  and le-rexp (Star r) (Star s) le-rexp (Star s) t
  thus le-rexp (Star r) t by (cases t) auto
next
  fix r s1 s2 t assume le-rexp (Plus s1 s2) t thus le-rexp (Star r) t by (cases t)
  auto
next
  fix r s1 s2 t assume le-rexp (Times s1 s2) t thus le-rexp (Star r) t by (cases
  t) auto
next
  fix r1 r2 s1 s2 t
  assume  $\bigwedge t. r1 = s1 \implies le-rexp r2 s2 \implies le-rexp s2 t \implies le-rexp r2 t$ 
   $\bigwedge t. r1 \neq s1 \implies le-rexp r1 s1 \implies le-rexp s1 t \implies le-rexp r1 t$ 
  le-rexp (Plus r1 r2) (Plus s1 s2) le-rexp (Plus s1 s2) t
  thus le-rexp (Plus r1 r2) t by (cases t) (auto split: if-split-asm intro: le-rexp-antisym)
next
  fix r1 r2 s1 s2 t assume le-rexp (Times s1 s2) t thus le-rexp (Plus r1 r2) t by
  (cases t) auto
next
  fix r1 r2 s1 s2 t
  assume  $\bigwedge t. r1 = s1 \implies le-rexp r2 s2 \implies le-rexp s2 t \implies le-rexp r2 t$ 
   $\bigwedge t. r1 \neq s1 \implies le-rexp r1 s1 \implies le-rexp s1 t \implies le-rexp r1 t$ 
  le-rexp (Times r1 r2) (Times s1 s2) le-rexp (Times s1 s2) t
  thus le-rexp (Times r1 r2) t by (cases t) (auto split: if-split-asm intro: le-rexp-antisym)
qed auto

instance proof
qed (auto simp add: less-eq-rexp-def less-rexp-def
  intro: le-rexp-refl le-rexp-antisym le-rexp-trans)

end

```

```

instantiation rexp :: (linorder) {linorder}
begin

lemma le-rexp-total: le-rexp (r :: 'a :: linorder rexp) s ∨ le-rexp s r
by (induction r s rule: le-rexp.induct) auto

instance proof
qed (unfold less-eq-rexp-def less-rexp-def, rule le-rexp-total)

end

end

```

### 3 Normalizing Derivative

```

theory NDerivative
imports
  Regular-Exp
begin

```

#### 3.1 Normalizing operations

associativity, commutativity, idempotence, zero

```

fun nPlus :: 'a::order rexp ⇒ 'a rexp ⇒ 'a rexp
where
  nPlus Zero r = r
| nPlus r Zero = r
| nPlus (Plus r s) t = nPlus r (nPlus s t)
| nPlus r (Plus s t) =
  (if r = s then (Plus s t)
   else if le-rexp r s then Plus r (Plus s t)
   else Plus s (nPlus r t))
| nPlus r s =
  (if r = s then r
   else if le-rexp r s then Plus r s
   else Plus s r)

```

```

lemma lang-nPlus[simp]: lang (nPlus r s) = lang (Plus r s)
by (induction r s rule: nPlus.induct) auto

```

associativity, zero, one

```

fun nTimes :: 'a::order rexp ⇒ 'a rexp ⇒ 'a rexp
where
  nTimes Zero - = Zero
| nTimes - Zero = Zero
| nTimes One r = r
| nTimes r One = r
| nTimes (Times r s) t = Times r (nTimes s t)

```



|  $nTimes\ r\ s = Times\ r\ s$

**lemma** *lang-nTimes[simp]*:  $lang\ (nTimes\ r\ s) = lang\ (Times\ r\ s)$   
**by** (*induction r s rule: nTimes.induct*) (*auto simp: conc-assoc*)

**primrec** *norm* :: 'a::order *rexp*  $\Rightarrow$  'a *rexp*

**where**

*norm Zero* = *Zero*  
| *norm One* = *One*  
| *norm (Atom a)* = *Atom a*  
| *norm (Plus r s)* = *nPlus (norm r) (norm s)*  
| *norm (Times r s)* = *nTimes (norm r) (norm s)*  
| *norm (Star r)* = *Star (norm r)*

**lemma** *lang-norm[simp]*:  $lang\ (norm\ r) = lang\ r$   
**by** (*induct r*) *auto*

**primrec** *nderiv* :: 'a::order  $\Rightarrow$  'a *rexp*  $\Rightarrow$  'a *rexp*

**where**

*nderiv - Zero* = *Zero*  
| *nderiv - One* = *Zero*  
| *nderiv a (Atom b)* = (*if a = b then One else Zero*)  
| *nderiv a (Plus r s)* = *nPlus (nderiv a r) (nderiv a s)*  
| *nderiv a (Times r s)* =  
    (*let r's = nTimes (nderiv a r) s*  
      *in if nullable r then nPlus r's (nderiv a s) else r's*)  
| *nderiv a (Star r)* = *nTimes (nderiv a r) (Star r)*

**lemma** *lang-nderiv*:  $lang\ (nderiv\ a\ r) = Deriv\ a\ (lang\ r)$   
**by** (*induction r*) (*auto simp: Let-def nullable-iff*)

**lemma** *deriv-no-occurrence*:

$x \notin atoms\ r \implies nderiv\ x\ r = Zero$   
**by** (*induction r*) *auto*

**lemma** *atoms-nPlus[simp]*:  $atoms\ (nPlus\ r\ s) = atoms\ r \cup atoms\ s$   
**by** (*induction r s rule: nPlus.induct*) *auto*

**lemma** *atoms-nTimes*:  $atoms\ (nTimes\ r\ s) \subseteq atoms\ r \cup atoms\ s$   
**by** (*induction r s rule: nTimes.induct*) *auto*

**lemma** *atoms-norm*:  $atoms\ (norm\ r) \subseteq atoms\ r$   
**by** (*induction r*) (*auto dest!: subsetD[OF atoms-nTimes]*)

**lemma** *atoms-nderiv*:  $atoms\ (nderiv\ a\ r) \subseteq atoms\ r$   
**by** (*induction r*) (*auto simp: Let-def dest!: subsetD[OF atoms-nTimes]*)

**end**

## 4 Deciding Regular Expression Equivalence

```

theory Equivalence-Checking
imports
  NDerivative
  HOL-Library.While-Combinator
begin

```

### 4.1 Bisimulation between languages and regular expressions

```

coinductive bisimilar :: 'a lang  $\Rightarrow$  'a lang  $\Rightarrow$  bool where
( $\square \in K \longleftrightarrow \square \in L$ )
 $\implies (\bigwedge x. \text{bisimilar } (\text{Deriv } x K) (\text{Deriv } x L))$ 
 $\implies \text{bisimilar } K L$ 

```

**lemma** *equal-if-bisimilar*:

**assumes** *bisimilar*  $K L$  **shows**  $K = L$

**proof** (*rule set-eqI*)

**fix**  $w$

**from**  $\langle \text{bisimilar } K L \rangle$  **show**  $w \in K \longleftrightarrow w \in L$

**proof** (*induct w arbitrary: K L*)

**case** *Nil* **thus** ?*case* **by** (*auto elim: bisimilar.cases*)

**next**

**case** (*Cons*  $a w K L$ )

**from**  $\langle \text{bisimilar } K L \rangle$  **have** *bisimilar* (*Deriv*  $a K$ ) (*Deriv*  $a L$ )

**by** (*auto elim: bisimilar.cases*)

**then have**  $w \in \text{Deriv } a K \longleftrightarrow w \in \text{Deriv } a L$  **by** (*rule Cons(1)*)

**thus** ?*case* **by** (*auto simp: Deriv-def*)

**qed**

**qed**

**lemma** *language-coinduct*:

**fixes**  $R$  (*infixl*  $\sim$  50)

**assumes**  $K \sim L$

**assumes**  $\bigwedge K L. K \sim L \implies (\square \in K \longleftrightarrow \square \in L)$

**assumes**  $\bigwedge K L x. K \sim L \implies \text{Deriv } x K \sim \text{Deriv } x L$

**shows**  $K = L$

**apply** (*rule equal-if-bisimilar*)

**apply** (*rule bisimilar.coinduct[of R, OF  $\langle K \sim L \rangle$ ]*)

**apply** (*auto simp: assms*)

**done**

**type-synonym** 'a *rexp-pair* = 'a *rexp* \* 'a *rexp*

**type-synonym** 'a *rexp-pairs* = 'a *rexp-pair* list

**definition** *is-bisimulation* :: 'a::order list  $\Rightarrow$  'a *rexp-pair* set  $\Rightarrow$  bool

**where**

*is-bisimulation* as  $R =$

$(\forall (r,s) \in R. (\text{atoms } r \cup \text{atoms } s \subseteq \text{set } as) \wedge (\text{nullable } r \longleftrightarrow \text{nullable } s) \wedge$   
 $(\forall a \in \text{set } as. (\text{nderiv } a r, \text{nderiv } a s) \in R))$

```

lemma bisim-lang-eq:
  assumes bisim: is-bisimulation as ps
  assumes (r, s) ∈ ps
  shows lang r = lang s
  proof -
    define ps' where ps' = insert (Zero, Zero) ps
    from bisim have bisim': is-bisimulation as ps'
      by (auto simp: ps'-def is-bisimulation-def)
    let ?R = λK L. (∃(r,s)∈ps'. K = lang r ∧ L = lang s)
    show ?thesis
    proof (rule language-coinduct[where R=?R])
      from ⟨(r, s) ∈ ps⟩
      have (r, s) ∈ ps' by (auto simp: ps'-def)
      thus ?R (lang r) (lang s) by auto
    next
      fix K L assume ?R K L
      then obtain r s where rs: (r, s) ∈ ps'
        and KL: K = lang r L = lang s by auto
      with bisim' have nullable r ⟷ nullable s
        by (auto simp: is-bisimulation-def)
      thus [] ∈ K ⟷ [] ∈ L by (auto simp: nullable-iff KL)
      fix a
      show ?R (Deriv a K) (Deriv a L)
      proof cases
        assume a ∈ set as
        with rs bisim'
        have (nderiv a r, nderiv a s) ∈ ps'
          by (auto simp: is-bisimulation-def)
        thus ?thesis by (force simp: KL lang-nderiv)
      next
        assume a ∉ set as
        with bisim' rs
        have a ∉ atoms r a ∉ atoms s by (auto simp: is-bisimulation-def)
        then have nderiv a r = Zero nderiv a s = Zero
          by (auto intro: deriv-no-occurrence)
        then have Deriv a K = lang Zero
          Deriv a L = lang Zero
          unfolding KL lang-nderiv[symmetric] by auto
        thus ?thesis by (auto simp: ps'-def)
      qed
    qed
  qed

```

## 4.2 Closure computation

**definition** closure ::

'a::order list ⇒ 'a rexp-pair ⇒ ('a rexp-pairs \* 'a rexp-pair set) option  
 where

*closure as* = *rtrancl-while* ( $\% (r,s). \text{nullable } r = \text{nullable } s$ )  
( $\% (r,s). \text{map } (\lambda a. (\text{nderiv } a \ r, \text{nderiv } a \ s)) \text{ as}$ )

**definition** *pre-bisim* :: 'a::order list  $\Rightarrow$  'a rexp  $\Rightarrow$  'a rexp  $\Rightarrow$   
'a rexp-pairs \* 'a rexp-pair set  $\Rightarrow$  bool

**where**

*pre-bisim as r s* = ( $\lambda (ws,R).$   
 $(r,s) \in R \wedge \text{set } ws \subseteq R \wedge$   
 $(\forall (r,s) \in R. \text{atoms } r \cup \text{atoms } s \subseteq \text{set } as) \wedge$   
 $(\forall (r,s) \in R - \text{set } ws. (\text{nullable } r \longleftrightarrow \text{nullable } s) \wedge$   
 $(\forall a \in \text{set } as. (\text{nderiv } a \ r, \text{nderiv } a \ s) \in R))$ )

**theorem** *closure-sound*:

**assumes** *result*: *closure as* ( $r,s$ ) = *Some*( $[],R$ )

**and** *atoms*:  $\text{atoms } r \cup \text{atoms } s \subseteq \text{set } as$

**shows**  $\text{lang } r = \text{lang } s$

**proof** –

**let** *?test* = *While-Combinator.rtrancl-while-test* ( $\% (r,s). \text{nullable } r = \text{nullable } s$ )  
**let** *?step* = *While-Combinator.rtrancl-while-step* ( $\% (r,s). \text{map } (\lambda a. (\text{nderiv } a \ r,$   
 $\text{nderiv } a \ s)) \text{ as}$ )

{ **fix** *st* **assume** *inv*: *pre-bisim as r s st* **and** *test*: *?test st*

**have** *pre-bisim as r s* (*?step st*)

**proof** (*cases st*)

**fix** *ws R* **assume** *st* = (*ws, R*)

**with** *test* **obtain** *r s t* **where** *st*: *st* = ( $(r, s) \# t, R$ ) **and** *nullable r* =  
*nullable s*

**by** (*cases ws*) *auto*

**with** *inv* **show** *?thesis* **using** *atoms-nderiv[of - r]* *atoms-nderiv[of - s]*

**unfolding** *st rtrancl-while-test.simps rtrancl-while-step.simps pre-bisim-def*

*Ball-def*

**by** *simp-all blast+*

**qed**

}

**moreover**

**from** *atoms*

**have** *pre-bisim as r s* ( $[(r,s)],\{(r,s)\}$ ) **by** (*simp add: pre-bisim-def*)

**ultimately** **have** *pre-bisim-ps*: *pre-bisim as r s* ( $[],R$ )

**by** (*rule while-option-rule[OF - result[unfolded closure-def rtrancl-while-def]]*)

**then** **have** *is-bisimulation as R* ( $r, s$ )  $\in R$

**by** (*auto simp: pre-bisim-def is-bisimulation-def*)

**thus**  $\text{lang } r = \text{lang } s$  **by** (*rule bisim-lang-eq*)

**qed**

### 4.3 Bisimulation-free proof of closure computation

The equivalence check can be viewed as the product construction of two automata. The state space is the reflexive transitive closure of the pair of next-state functions, i.e. derivatives.

**lemma** *rtrancl-nderiv-nderivs*: **defines** *nderivs* == *foldl* ( $\% r \ a. \text{nderiv } a \ r$ )

**shows**  $\{((r,s),(nderiv\ a\ r,nderiv\ a\ s))\mid r\ s\ a.\ a : A\}^{\widehat{*}} =$   
 $\{((r,s),(nderivs\ r\ w,nderivs\ s\ w))\mid r\ s\ w.\ w : lists\ A\}$  (**is**  $?L = ?R$ )

**proof** –

**note**  $[simp] = nderivs-def$

**{ fix**  $r\ s\ r'\ s'$

**have**  $((r,s),(r',s')) : ?L \implies ((r,s),(r',s')) : ?R$

**proof**(*induction rule: converse-rtrancl-induct2*)

**case refl show**  $?case$  **by** (*force intro!: foldl.simps(1)[symmetric]*)

**next**

**case step thus**  $?case$  **by**(*force intro!: foldl.simps(2)[symmetric]*)

**qed**

**} moreover**

**{ fix**  $r\ s\ r'\ s'$

**{ fix**  $w$  **have**  $\forall x \in set\ w.\ x \in A \implies ((r,s), nderivs\ r\ w, nderivs\ s\ w) : ?L$

**proof**(*induction w rule: rev-induct*)

**case Nil show**  $?case$  **by** *simp*

**next**

**case snoc thus**  $?case$  **by** (*auto elim!: rtrancl-into-rtrancl*)

**qed**

**}**

**hence**  $((r,s),(r',s')) : ?R \implies ((r,s),(r',s')) : ?L$  **by** *auto*

**} ultimately show**  $?thesis$  **by** (*auto simp: in-lists-conv-set*) **blast**

**qed**

**lemma** *nullable-nderivs:*

*nullable (foldl (%r a. nderiv a r) r w) = (w : lang r)*

**by** (*induct w arbitrary: r*) (*simp-all add: nullable-iff lang-nderiv Deriv-def*)

**theorem** *closure-sound-complete:*

**assumes** *result: closure as (r,s) = Some(ws,R)*

**and** *atoms: set as = atoms r  $\cup$  atoms s*

**shows**  $ws = [] \iff lang\ r = lang\ s$

**proof** –

**have** *leq: (lang r = lang s) =*

$(\forall (r',s') \in \{((r0,s0),(nderiv\ a\ r0,nderiv\ a\ s0))\mid r0\ s0\ a.\ a : set\ as\}^{\widehat{*}} \text{ “ } \{(r,s)\}.$   
*nullable r' = nullable s')*

**by**(*simp add: atoms rtrancl-nderiv-nderivs Ball-def lang-eq-ext imp-ex nullable-nderivs*

*del:Un-iff*)

**have**  $\{(x,y).\ y \in set\ ((\lambda(p,q).\ map\ (\lambda a.\ (nderiv\ a\ p,\ nderiv\ a\ q))\ as)\ x)\} =$   
 $\{((r,s), nderiv\ a\ r,\ nderiv\ a\ s) \mid r\ s\ a.\ a \in set\ as\}$

**by** *auto*

**with** *atoms rtrancl-while-Some[OF result[unfolded closure-def]]*

**show**  $?thesis$  **by** (*auto simp add: leq Ball-def split: if-splits*)

**qed**

#### 4.4 The overall procedure

*primrec add-atoms :: 'a rexp  $\Rightarrow$  'a list  $\Rightarrow$  'a list*

**where**

```
  add-atoms Zero = id
| add-atoms One = id
| add-atoms (Atom a) = List.insert a
| add-atoms (Plus r s) = add-atoms s o add-atoms r
| add-atoms (Times r s) = add-atoms s o add-atoms r
| add-atoms (Star r) = add-atoms r
```

**lemma** *set-add-atoms*:  $set (add-atoms r as) = atoms r \cup set as$   
**by** (*induct r arbitrary: as*) *auto*

**definition** *check-eqv* ::  $nat \text{ rexp} \Rightarrow nat \text{ rexp} \Rightarrow bool$  **where**

```
check-eqv r s =
  (let nr = norm r; ns = norm s; as = add-atoms nr (add-atoms ns [])
   in case closure as (nr, ns) of
     Some([],-)  $\Rightarrow$  True | -  $\Rightarrow$  False)
```

**lemma** *soundness*:

**assumes** *check-eqv r s* **shows**  $lang r = lang s$

**proof** –

```
  let ?nr = norm r let ?ns = norm s
  let ?as = add-atoms ?nr (add-atoms ?ns [])
  obtain R where 1: closure ?as (?nr, ?ns) = Some([],R)
    using assms by (auto simp: check-eqv-def Let-def split:option.splits list.splits)
  then have  $lang (norm r) = lang (norm s)$ 
    by (rule closure-sound) (auto simp: set-add-atoms dest!: subsetD[OF atoms-norm])
  thus  $lang r = lang s$  by simp
```

**qed**

Test:

**lemma** *check-eqv (Plus One (Times (Atom 0) (Star(Atom 0)))) (Star(Atom 0))*  
**by** *eval*

**end**

## 5 Regular Expressions as Homogeneous Binary Relations

**theory** *Relation-Interpretation*

**imports** *Regular-Exp*

**begin**

**primrec** *rel* ::  $('a \Rightarrow ('b * 'b) set) \Rightarrow 'a \text{ rexp} \Rightarrow ('b * 'b) set$

**where**

```
rel v Zero = {} |
rel v One = Id |
rel v (Atom a) = v a |
```

$rel\ v\ (Plus\ r\ s) = rel\ v\ r \cup rel\ v\ s \mid$   
 $rel\ v\ (Times\ r\ s) = rel\ v\ r\ O\ rel\ v\ s \mid$   
 $rel\ v\ (Star\ r) = (rel\ v\ r)^{\hat{*}}$

**primrec** *word-rel* :: ('a  $\Rightarrow$  ('b \* 'b) set)  $\Rightarrow$  'a list  $\Rightarrow$  ('b \* 'b) set  
**where**  
*word-rel* v [] = *Id*  
 $\mid$  *word-rel* v (a#as) = v a O *word-rel* v as

**lemma** *word-rel-append*:  
 $word\text{-}rel\ v\ w\ O\ word\text{-}rel\ v\ w' = word\text{-}rel\ v\ (w\ @\ w')$   
**by** (*rule sym*) (*induct w, auto*)

**lemma** *rel-word-rel*:  $rel\ v\ r = (\bigcup_{w \in lang\ r} word\text{-}rel\ v\ w)$

**proof** (*induct r*)  
**case** *Times* **thus** ?*case*  
**by** (*auto simp: rel-def word-rel-append conc-def relcomp-UNION-distrib rel-comp-UNION-distrib2*)  
**next**  
**case** (*Star r*)  
**{** **fix** *n*  
**have**  $(rel\ v\ r)^{\hat{\sim} n} = (\bigcup_{w \in lang\ r^{\hat{\sim} n}} word\text{-}rel\ v\ w)$   
**proof** (*induct n*)  
**case** 0 **show** ?*case* **by** *simp*  
**next**  
**case** (*Suc n*) **thus** ?*case*  
**unfolding** *relpow.simps relpow-commute[symmetric]*  
**by** (*auto simp add: Star conc-def word-rel-append relcomp-UNION-distrib relcomp-UNION-distrib2*)  
**qed** **}**  
**thus** ?*case* **unfolding** *rel.simps*  
**by** (*force simp: rtrancl-power star-def*)  
**qed** *auto*

Soundness:

**lemma** *soundness*:  
 $lang\ r = lang\ s \implies rel\ v\ r = rel\ v\ s$   
**unfolding** *rel-word-rel* **by** *auto*

**end**

## 6 Proving Relation (In)equalities via Regular Expressions

**theory** *Regexp-Method*  
**imports** *Equivalence-Checking Relation-Interpretation*  
**begin**

**primrec** *rel-of-regex* :: ('a \* 'a) set list  $\Rightarrow$  nat rexp  $\Rightarrow$  ('a \* 'a) set **where**  
*rel-of-regex* vs Zero = {} |  
*rel-of-regex* vs One = Id |  
*rel-of-regex* vs (Atom i) = vs ! i |  
*rel-of-regex* vs (Plus r s) = *rel-of-regex* vs r  $\cup$  *rel-of-regex* vs s |  
*rel-of-regex* vs (Times r s) = *rel-of-regex* vs r O *rel-of-regex* vs s |  
*rel-of-regex* vs (Star r) = (*rel-of-regex* vs r)<sup>\*</sup>

**lemma** *rel-of-regex-rel*: *rel-of-regex* vs r = rel ( $\lambda i. vs ! i$ ) r  
**by** (induct r) auto

**primrec** *rel-eq* **where**  
*rel-eq* (r, s) vs = (*rel-of-regex* vs r = *rel-of-regex* vs s)

**lemma** *rel-eqI*: *check-req* r s  $\Longrightarrow$  *rel-eq* (r, s) vs  
**unfolding** *rel-eq.simps* *rel-of-regex-rel*  
**by** (rule *Relation-Interpretation.soundness*)  
(rule *Equivalence-Checking.soundness*)

**lemmas** *regex-reify* = *rel-of-regex.simps* *rel-eq.simps*  
**lemmas** *regex-unfold* = *tranc1-unfold-left* *subset-Un-eq*

**ML**  $\langle$   
*local*

*fun* *check-req* (ct, b) = *Thm.mk-binop* @{*cterm* *Pure.eq* :: bool  $\Rightarrow$  bool  $\Rightarrow$  prop}  
*ct* (if b then @{*cterm* *True*} else @{*cterm* *False*});

*val* (-, *check-req-oracle*) = *Context.>>>* (*Context.map-theory-result*  
(*Thm.add-oracle* (@{*binding* *check-req*}, *check-req*)));

*in*

*val* *regex-conv* =  
@{*computation-conv* bool terms: *check-req* datatypes: nat rexp}  
(fn - => fn b => fn ct => *check-req-oracle* (ct, b))

*end*  
 $\rangle$

**method-setup** *regex* =  $\langle$   
*Scan.succeed* (fn *ctxt* =>  
SIMPLE-METHOD' (  
(TRY o *eresolve-tac* *ctxt* @{*thms* *rev-subsetD*})  
THEN' (*Subgoal.FOCUS-PARAMS* (fn {*context* = *ctxt'*, ...} =>  
TRY (*Local-Defs.unfold-tac* *ctxt'* @{*thms* *regex-unfold*})  
THEN *Reification.tac* *ctxt'* @{*thms* *regex-reify*} NONE 1  
THEN *resolve-tac* *ctxt'* @{*thms* *rel-eqI*} 1



```

    THEN CONVERSION (HOLogic.Trueprop-conv (regexp-conv ctxt') 1
    THEN resolve-tac ctxt' [TrueI] 1) ctxt)))
  › ⟨decide relation equalities via regular expressions⟩

```

```

hide-const (open) le-regex nPlus nTimes norm nullable bisimilar is-bisimulation
closure
pre-bisim add-atoms check-eqv rel word-rel rel-eq

```

Example:

```

lemma  $(r \cup s^{\wedge+})^{\wedge*} = (r \cup s)^{\wedge*}$ 
by regexp

```

**end**

## 7 Basic constructions on regular expressions

```

theory Regexp-Constructions
imports
  Main
  HOL-Library.Sublist
  Regular-Exp
begin

```

### 7.1 Reverse language

```

lemma rev-conc [simp]:  $rev \text{ ' } (A @@ B) = rev \text{ ' } B @@ rev \text{ ' } A$ 
unfolding conc-def image-def by force

```

```

lemma rev-compower [simp]:  $rev \text{ ' } (A \text{ } \sim n) = (rev \text{ ' } A) \text{ } \sim n$ 
by (induction n) (simp-all add: conc-pow-comm)

```

```

lemma rev-star [simp]:  $rev \text{ ' } star A = star (rev \text{ ' } A)$ 
by (simp add: star-def image-UN)

```

### 7.2 Substituting characters in a language

```

definition subst-word :: ('a  $\Rightarrow$  'b list)  $\Rightarrow$  'a list  $\Rightarrow$  'b list where
  subst-word f xs = concat (map f xs)

```

```

lemma subst-word-Nil [simp]:  $subst-word f [] = []$ 
by (simp add: subst-word-def)

```

```

lemma subst-word-singleton [simp]:  $subst-word f [x] = f x$ 
by (simp add: subst-word-def)

```

```

lemma subst-word-append [simp]:  $subst-word f (xs @ ys) = subst-word f xs @$ 
 $subst-word f ys$ 
by (simp add: subst-word-def)

```

**lemma** *subst-word-Cons* [simp]:  $\text{subst-word } f (x \# xs) = f x @ \text{subst-word } f xs$   
**by** (*simp add: subst-word-def*)

**lemma** *subst-word-conc* [simp]:  $\text{subst-word } f ' (A @@ B) = \text{subst-word } f ' A @@ \text{subst-word } f ' B$   
**unfolding** *conc-def image-def* **by** *force*

**lemma** *subst-word-compower* [simp]:  $\text{subst-word } f ' (A \overset{\sim}{\sim} n) = (\text{subst-word } f ' A) \overset{\sim}{\sim} n$   
**by** (*induction n simp-all*)

**lemma** *subst-word-star* [simp]:  $\text{subst-word } f ' (\text{star } A) = \text{star } (\text{subst-word } f ' A)$   
**by** (*simp add: star-def image-UN*)

Suffix language

**definition** *Suffixes* :: 'a list set  $\Rightarrow$  'a list set **where**  
*Suffixes* A = {w.  $\exists q. q @ w \in A$ }

**lemma** *Suffixes-altdef* [code]:  $\text{Suffixes } A = (\bigcup w \in A. \text{set } (\text{suffixes } w))$   
**unfolding** *Suffixes-def set-suffixes-eq suffix-def* **by** *blast*

**lemma** *Nil-in-Suffixes-iff* [simp]:  $\emptyset \in \text{Suffixes } A \longleftrightarrow A \neq \{\}$   
**by** (*auto simp: Suffixes-def*)

**lemma** *Suffixes-empty* [simp]:  $\text{Suffixes } \{\} = \{\}$   
**by** (*auto simp: Suffixes-def*)

**lemma** *Suffixes-empty-iff* [simp]:  $\text{Suffixes } A = \{\} \longleftrightarrow A = \{\}$   
**by** (*auto simp: Suffixes-altdef*)

**lemma** *Suffixes-singleton* [simp]:  $\text{Suffixes } \{xs\} = \text{set } (\text{suffixes } xs)$   
**by** (*auto simp: Suffixes-altdef*)

**lemma** *Suffixes-insert*:  $\text{Suffixes } (\text{insert } xs A) = \text{set } (\text{suffixes } xs) \cup \text{Suffixes } A$   
**by** (*simp add: Suffixes-altdef*)

**lemma** *Suffixes-conc* [simp]:  $A \neq \{\} \implies \text{Suffixes } (A @@ B) = \text{Suffixes } B \cup (\text{Suffixes } A @@ B)$   
**unfolding** *Suffixes-altdef conc-def* **by** (*force simp: suffix-append*)

**lemma** *Suffixes-union* [simp]:  $\text{Suffixes } (A \cup B) = \text{Suffixes } A \cup \text{Suffixes } B$   
**by** (*auto simp: Suffixes-def*)

**lemma** *Suffixes-UNION* [simp]:  $\text{Suffixes } (\bigcup (f ' A)) = \bigcup ((\lambda x. \text{Suffixes } (f x)) ' A)$   
**by** (*auto simp: Suffixes-def*)

**lemma** *Suffixes-compower*:  
**assumes**  $A \neq \{\}$   
**shows**  $\text{Suffixes } (A \overset{\sim}{\sim} n) = \text{insert } \emptyset (\text{Suffixes } A @@ (\bigcup_{k < n. A \overset{\sim}{\sim} k}))$

**proof** (*induction n*)  
**case** (*Suc n*)  
**from** *Suc* **have** *Suffixes* ( $A \overset{\sim}{\sim} \text{Suc } n$ ) =  
 $\text{insert } [] (\text{Suffixes } A \text{ @@ } ((\bigcup_{k < n} A \overset{\sim}{\sim} k) \cup A \overset{\sim}{\sim} n))$   
**by** (*simp-all add: assms conc-Un-distrib*)  
**also have**  $(\bigcup_{k < n} A \overset{\sim}{\sim} k) \cup A \overset{\sim}{\sim} n = (\bigcup_{k \in \text{insert } n \{..<n\}} A \overset{\sim}{\sim} k)$  **by** *blast*  
**also have**  $\text{insert } n \{..<n\} = \{..<\text{Suc } n\}$  **by** *auto*  
**finally show** *?case* .  
**qed** *simp-all*

**lemma** *Suffixes-star* [*simp*]:  
**assumes**  $A \neq \{\}$   
**shows**  $\text{Suffixes } (\text{star } A) = \text{Suffixes } A \text{ @@ } \text{star } A$   
**proof** –  
**have**  $\text{star } A = (\bigcup n. A \overset{\sim}{\sim} n)$  **unfolding** *star-def* ..  
**also have**  $\text{Suffixes } \dots = (\bigcup x. \text{Suffixes } (A \overset{\sim}{\sim} x))$  **by** *simp*  
**also have**  $\dots = (\bigcup n. \text{insert } [] (\text{Suffixes } A \text{ @@ } (\bigcup_{k < n} A \overset{\sim}{\sim} k)))$   
**using** *assms* **by** (*subst Suffixes-compower*) *auto*  
**also have**  $\dots = \text{insert } [] (\text{Suffixes } A \text{ @@ } (\bigcup n. (\bigcup_{k < n} A \overset{\sim}{\sim} k)))$   
**by** (*simp-all add: conc-UNION-distrib*)  
**also have**  $(\bigcup n. (\bigcup_{k < n} A \overset{\sim}{\sim} k)) = (\bigcup n. A \overset{\sim}{\sim} n)$  **by** *auto*  
**also have**  $\dots = \text{star } A$  **unfolding** *star-def* ..  
**also have**  $\text{insert } [] (\text{Suffixes } A \text{ @@ } \text{star } A) = \text{Suffixes } A \text{ @@ } \text{star } A$   
**using** *assms* **by** *auto*  
**finally show** *?thesis* .  
**qed**

Prefix language

**definition** *Prefixes* :: 'a list set  $\Rightarrow$  'a list set **where**  
 $\text{Prefixes } A = \{w. \exists q. w @ q \in A\}$

**lemma** *Prefixes-altdef* [*code*]:  $\text{Prefixes } A = (\bigcup w \in A. \text{set } (\text{prefixes } w))$   
**unfolding** *Prefixes-def set-prefixes-eq prefix-def* **by** *blast*

**lemma** *Nil-in-Prefixes-iff* [*simp*]:  $[] \in \text{Prefixes } A \iff A \neq \{\}$   
**by** (*auto simp: Prefixes-def*)

**lemma** *Prefixes-empty* [*simp*]:  $\text{Prefixes } \{\} = \{\}$   
**by** (*auto simp: Prefixes-def*)

**lemma** *Prefixes-empty-iff* [*simp*]:  $\text{Prefixes } A = \{\} \iff A = \{\}$   
**by** (*auto simp: Prefixes-altdef*)

**lemma** *Prefixes-singleton* [*simp*]:  $\text{Prefixes } \{xs\} = \text{set } (\text{prefixes } xs)$   
**by** (*auto simp: Prefixes-altdef*)

**lemma** *Prefixes-insert*:  $\text{Prefixes } (\text{insert } xs A) = \text{set } (\text{prefixes } xs) \cup \text{Prefixes } A$   
**by** (*simp add: Prefixes-altdef*)

**lemma** *Prefixes-conc* [*simp*]:  $B \neq \{\}$   $\implies$   $\text{Prefixes } (A \text{ @@ } B) = \text{Prefixes } A \cup (A \text{ @@ } \text{Prefixes } B)$

**unfolding** *Prefixes-altdef conc-def* **by** (*force simp: prefix-append*)

**lemma** *Prefixes-union* [*simp*]:  $\text{Prefixes } (A \cup B) = \text{Prefixes } A \cup \text{Prefixes } B$   
**by** (*auto simp: Prefixes-def*)

**lemma** *Prefixes-UNION* [*simp*]:  $\text{Prefixes } (\bigcup (f \text{ ' } A)) = \bigcup ((\lambda x. \text{Prefixes } (f x)) \text{ ' } A)$   
**by** (*auto simp: Prefixes-def*)

**lemma** *Prefixes-rev*:  $\text{Prefixes } (\text{rev ' } A) = \text{rev ' } \text{Suffixes } A$   
**by** (*auto simp: Prefixes-altdef prefixes-rev Suffixes-altdef*)

**lemma** *Suffixes-rev*:  $\text{Suffixes } (\text{rev ' } A) = \text{rev ' } \text{Prefixes } A$   
**by** (*auto simp: Prefixes-altdef suffixes-rev Suffixes-altdef*)

**lemma** *Prefixes-compower*:

**assumes**  $A \neq \{\}$

**shows**  $\text{Prefixes } (A \text{ } \sim n) = \text{insert } [] ((\bigcup_{k < n}. A \text{ } \sim k) \text{ @@ } \text{Prefixes } A)$

**proof** –

**have**  $A \text{ } \sim n = \text{rev ' } ((\text{rev ' } A) \text{ } \sim n)$  **by** (*simp add: image-image*)

**also have**  $\text{Prefixes } \dots = \text{insert } [] ((\bigcup_{k < n}. A \text{ } \sim k) \text{ @@ } \text{Prefixes } A)$

**unfolding** *Prefixes-rev*

**by** (*subst Suffixes-compower*) (*simp-all add: image-UN image-image Suffixes-rev assms*)

**finally show** *?thesis* .

**qed**

**lemma** *Prefixes-star* [*simp*]:

**assumes**  $A \neq \{\}$

**shows**  $\text{Prefixes } (\text{star } A) = \text{star } A \text{ @@ } \text{Prefixes } A$

**proof** –

**have**  $\text{star } A = \text{rev ' } \text{star } (\text{rev ' } A)$  **by** (*simp add: image-image*)

**also have**  $\text{Prefixes } \dots = \text{star } A \text{ @@ } \text{Prefixes } A$

**unfolding** *Prefixes-rev*

**by** (*subst Suffixes-star*) (*simp-all add: assms image-image Suffixes-rev*)

**finally show** *?thesis* .

**qed**

### 7.3 Subword language

The language of all sub-words, i.e. all words that are a contiguous sublist of a word in the original language.

**definition** *Sublists* :: 'a list set  $\Rightarrow$  'a list set **where**

$\text{Sublists } A = \{w. \exists q \in A. \text{sublist } w q\}$

**lemma** *Sublists-altdef* [*code*]:  $\text{Sublists } A = (\bigcup_{w \in A}. \text{set } (\text{sublists } w))$

**by** (*auto simp: Sublists-def*)

**lemma** *Sublists-empty* [*simp*]:  $\text{Sublists } \{\} = \{\}$   
**by** (*auto simp: Sublists-def*)

**lemma** *Sublists-singleton* [*simp*]:  $\text{Sublists } \{w\} = \text{set } (\text{sublists } w)$   
**by** (*auto simp: Sublists-altdef*)

**lemma** *Sublists-insert*:  $\text{Sublists } (\text{insert } w A) = \text{set } (\text{sublists } w) \cup \text{Sublists } A$   
**by** (*auto simp: Sublists-altdef*)

**lemma** *Sublists-Un* [*simp*]:  $\text{Sublists } (A \cup B) = \text{Sublists } A \cup \text{Sublists } B$   
**by** (*auto simp: Sublists-altdef*)

**lemma** *Sublists-UN* [*simp*]:  $\text{Sublists } (\bigcup (f \text{ ` } A)) = \bigcup ((\lambda x. \text{Sublists } (f x)) \text{ ` } A)$   
**by** (*auto simp: Sublists-altdef*)

**lemma** *Sublists-conv-Prefixes*:  $\text{Sublists } A = \text{Prefixes } (\text{Suffixes } A)$   
**by** (*auto simp: Sublists-def Prefixes-def Suffixes-def sublist-def*)

**lemma** *Sublists-conv-Suffixes*:  $\text{Sublists } A = \text{Suffixes } (\text{Prefixes } A)$   
**by** (*auto simp: Sublists-def Prefixes-def Suffixes-def sublist-def*)

**lemma** *Sublists-conc* [*simp*]:  
**assumes**  $A \neq \{\}$   $B \neq \{\}$   
**shows**  $\text{Sublists } (A @ B) = \text{Sublists } A \cup \text{Sublists } B \cup \text{Suffixes } A @ \text{Prefixes } B$   
**using** *assms unfolding Sublists-conv-Suffixes* **by** *auto*

**lemma** *star-not-empty* [*simp*]:  $\text{star } A \neq \{\}$   
**by** *auto*

**lemma** *Sublists-star*:  
 $A \neq \{\} \implies \text{Sublists } (\text{star } A) = \text{Sublists } A \cup \text{Suffixes } A @ \text{star } A @ \text{Prefixes } A$   
**by** (*simp add: Sublists-conv-Prefixes*)

**lemma** *Prefixes-subset-Sublists*:  $\text{Prefixes } A \subseteq \text{Sublists } A$   
**unfolding** *Prefixes-def Sublists-def* **by** *auto*

**lemma** *Suffixes-subset-Sublists*:  $\text{Suffixes } A \subseteq \text{Sublists } A$   
**unfolding** *Suffixes-def Sublists-def* **by** *auto*

## 7.4 Fragment language

The following is the fragment language of a given language, i.e. the set of all words that are (not necessarily contiguous) sub-sequences of a word in the original language.

**definition** *Subseqs* **where**  $\text{Subseqs } A = (\bigcup w \in A. \text{set } (\text{subseqs } w))$

**lemma** *Subseqs-empty* [simp]:  $\text{Subseqs } \{\} = \{\}$   
**by** (*simp add: Subseqs-def*)

**lemma** *Subseqs-insert* [simp]:  $\text{Subseqs } (\text{insert } xs \ A) = \text{set } (\text{subseqs } xs) \cup \text{Subseqs } A$   
**by** (*simp add: Subseqs-def*)

**lemma** *Subseqs-singleton* [simp]:  $\text{Subseqs } \{xs\} = \text{set } (\text{subseqs } xs)$   
**by** *simp*

**lemma** *Subseqs-Un* [simp]:  $\text{Subseqs } (A \cup B) = \text{Subseqs } A \cup \text{Subseqs } B$   
**by** (*simp add: Subseqs-def*)

**lemma** *Subseqs-UNION* [simp]:  $\text{Subseqs } (\bigcup (f \ ' A)) = \bigcup ((\lambda x. \text{Subseqs } (f \ x)) \ ' A)$   
**by** (*simp add: Subseqs-def*)

**lemma** *Subseqs-conc* [simp]:  $\text{Subseqs } (A \ @\@ B) = \text{Subseqs } A \ @\@ \text{Subseqs } B$   
**proof** *safe*

**fix** *xs* **assume**  $xs \in \text{Subseqs } (A \ @\@ B)$   
**then obtain** *ys zs* **where**  $ys \in A \ zs \in B \ \text{subseq } xs \ (ys \ @ \ zs)$   
**by** (*auto simp: Subseqs-def conc-def*)  
**from**  $*(3)$  **obtain** *xs1 xs2* **where**  $xs = xs1 \ @ \ xs2 \ \text{subseq } xs1 \ ys \ \text{subseq } xs2 \ zs$   
**by** (*rule subseq-appendE*)  
**with**  $*(1,2)$  **show**  $xs \in \text{Subseqs } A \ @\@ \text{Subseqs } B$  **by** (*auto simp: Subseqs-def set-subseqs-eq*)  
**next**  
**fix** *xs* **assume**  $xs \in \text{Subseqs } A \ @\@ \text{Subseqs } B$   
**then obtain** *xs1 xs2 ys zs*  
**where**  $xs = xs1 \ @ \ xs2 \ \text{subseq } xs1 \ ys \ \text{subseq } xs2 \ zs \ ys \in A \ zs \in B$   
**by** (*auto simp: conc-def Subseqs-def*)  
**thus**  $xs \in \text{Subseqs } (A \ @\@ B)$  **by** (*force simp: Subseqs-def conc-def intro: list-emb-append-mono*)  
**qed**

**lemma** *Subseqs-compower* [simp]:  $\text{Subseqs } (A \ \sim\sim n) = \text{Subseqs } A \ \sim\sim n$   
**by** (*induction n*) *simp-all*

**lemma** *Subseqs-star* [simp]:  $\text{Subseqs } (\text{star } A) = \text{star } (\text{Subseqs } A)$   
**by** (*simp add: star-def*)

**lemma** *Sublists-subset-Subseqs*:  $\text{Sublists } A \subseteq \text{Subseqs } A$   
**by** (*auto simp: Sublists-def Subseqs-def dest!: sublist-imp-subseq*)

## 7.5 Various regular expression constructions

A construction for language reversal of a regular expression:

**primrec** *rexp-rev* **where**  
*rexp-rev* *Zero* = *Zero*  
| *rexp-rev* *One* = *One*

|  $\text{rexp-rev (Atom } x) = \text{Atom } x$   
|  $\text{rexp-rev (Plus } r \ s) = \text{Plus (rexp-rev } r) \ (\text{rexp-rev } s)$   
|  $\text{rexp-rev (Times } r \ s) = \text{Times (rexp-rev } s) \ (\text{rexp-rev } r)$   
|  $\text{rexp-rev (Star } r) = \text{Star (rexp-rev } r)$

**lemma** *lang-rexp-rev* [simp]:  $\text{lang (rexp-rev } r) = \text{rev } \text{' lang } r$   
**by** (*induction*  $r$ ) (*simp-all* add: *image-Un*)

The obvious construction for a singleton-language regular expression:

**fun** *rexp-of-word* **where**  
*rexp-of-word* [] = *One*  
| *rexp-of-word* [x] = *Atom* x  
| *rexp-of-word* (x#xs) = *Times (Atom* x) (*rexp-of-word* xs)

**lemma** *lang-rexp-of-word* [simp]:  $\text{lang (rexp-of-word } xs) = \{xs\}$   
**by** (*induction*  $xs$  rule: *rexp-of-word.induct*) (*simp-all* add: *conc-def*)

**lemma** *size-rexp-of-word* [simp]:  $\text{size (rexp-of-word } xs) = \text{Suc } (2 * (\text{length } xs - 1))$   
**by** (*induction*  $xs$  rule: *rexp-of-word.induct*) *auto*

Character substitution in a regular expression:

**primrec** *rexp-subst* **where**  
*rexp-subst* f *Zero* = *Zero*  
| *rexp-subst* f *One* = *One*  
| *rexp-subst* f (*Atom* x) = *rexp-of-word* (f x)  
| *rexp-subst* f (*Plus* r s) = *Plus (rexp-subst* f r) (*rexp-subst* f s)  
| *rexp-subst* f (*Times* r s) = *Times (rexp-subst* f r) (*rexp-subst* f s)  
| *rexp-subst* f (*Star* r) = *Star (rexp-subst* f r)

**lemma** *lang-rexp-subst*:  $\text{lang (rexp-subst } f \ r) = \text{subst-word } f \ \text{' lang } r$   
**by** (*induction*  $r$ ) (*simp-all* add: *image-Un*)

Suffix language of a regular expression:

**primrec** *suffix-rexp* :: 'a *rexp*  $\Rightarrow$  'a *rexp* **where**  
*suffix-rexp* *Zero* = *Zero*  
| *suffix-rexp* *One* = *One*  
| *suffix-rexp* (*Atom* a) = *Plus (Atom* a) *One*  
| *suffix-rexp* (*Plus* r s) = *Plus (suffix-rexp* r) (*suffix-rexp* s)  
| *suffix-rexp* (*Times* r s) =  
    (*if rexp-empty* r then *Zero* else *Plus (Times (suffix-rexp* r) s) (*suffix-rexp* s))  
| *suffix-rexp* (*Star* r) =  
    (*if rexp-empty* r then *One* else *Times (suffix-rexp* r) (*Star* r))

**theorem** *lang-suffix-rexp* [simp]:  
 $\text{lang (suffix-rexp } r) = \text{Suffixes (lang } r)$   
**by** (*induction*  $r$ ) (*auto simp: rexp-empty-iff*)

Prefix language of a regular expression:

**primrec** *prefix-rexp* :: 'a *rexp*  $\Rightarrow$  'a *rexp* **where**

```

  prefix-regex Zero = Zero
| prefix-regex One = One
| prefix-regex (Atom a) = Plus (Atom a) One
| prefix-regex (Plus r s) = Plus (prefix-regex r) (prefix-regex s)
| prefix-regex (Times r s) =
  (if rexp-empty s then Zero else Plus (Times r (prefix-regex s)) (prefix-regex r))
| prefix-regex (Star r) =
  (if rexp-empty r then One else Times (Star r) (prefix-regex r))

```

**theorem** lang-prefix-regex [simp]:

```

lang (prefix-regex r) = Prefixes (lang r)
by (induction r) (auto simp: rexp-empty-iff)

```

Sub-word language of a regular expression

**primrec** sublist-regex :: 'a rexp  $\Rightarrow$  'a rexp **where**

```

  sublist-regex Zero = Zero
| sublist-regex One = One
| sublist-regex (Atom a) = Plus (Atom a) One
| sublist-regex (Plus r s) = Plus (sublist-regex r) (sublist-regex s)
| sublist-regex (Times r s) =
  (if rexp-empty r  $\vee$  rexp-empty s then Zero else
   Plus (sublist-regex r) (Plus (sublist-regex s) (Times (suffix-regex r) (prefix-regex
s))))
| sublist-regex (Star r) =
  (if rexp-empty r then One else
   Plus (sublist-regex r) (Times (suffix-regex r) (Times (Star r) (prefix-regex r))))

```

**theorem** lang-sublist-regex [simp]:

```

lang (sublist-regex r) = Sublists (lang r)
by (induction r) (auto simp: rexp-empty-iff Sublists-star)

```

Fragment language of a regular expression:

**primrec** subseqs-regex :: 'a rexp  $\Rightarrow$  'a rexp **where**

```

  subseqs-regex Zero = Zero
| subseqs-regex One = One
| subseqs-regex (Atom x) = Plus (Atom x) One
| subseqs-regex (Plus r s) = Plus (subseqs-regex r) (subseqs-regex s)
| subseqs-regex (Times r s) = Times (subseqs-regex r) (subseqs-regex s)
| subseqs-regex (Star r) = Star (subseqs-regex r)

```

**lemma** lang-subseqs-regex [simp]: lang (subseqs-regex r) = Subseqs (lang r)

```

by (induction r) auto

```

Subword language of a regular expression

**end**

## 8 Derivatives of regular expressions

**theory** Derivatives



**imports** *Regular-Exp*  
**begin**

This theory is based on work by Brozowski [2] and Antimirov [1].

## 8.1 Brzowski's derivatives of regular expressions

**fun**

*deriv* :: 'a  $\Rightarrow$  'a *rexp*  $\Rightarrow$  'a *rexp*

**where**

*deriv* *c* (*Zero*) = *Zero*

| *deriv* *c* (*One*) = *Zero*

| *deriv* *c* (*Atom* *c'*) = (if *c* = *c'* then *One* else *Zero*)

| *deriv* *c* (*Plus* *r1* *r2*) = *Plus* (*deriv* *c* *r1*) (*deriv* *c* *r2*)

| *deriv* *c* (*Times* *r1* *r2*) =

(if *nullable* *r1* then *Plus* (*Times* (*deriv* *c* *r1*) *r2*) (*deriv* *c* *r2*) else *Times* (*deriv* *c* *r1*) *r2*)

| *deriv* *c* (*Star* *r*) = *Times* (*deriv* *c* *r*) (*Star* *r*)

**fun**

*derivs* :: 'a *list*  $\Rightarrow$  'a *rexp*  $\Rightarrow$  'a *rexp*

**where**

*derivs* [] *r* = *r*

| *derivs* (*c* # *s*) *r* = *derivs* *s* (*deriv* *c* *r*)

**lemma** *atoms-deriv-subset*: *atoms* (*deriv* *x* *r*)  $\subseteq$  *atoms* *r*

**by** (*induction* *r*) (*auto*)

**lemma** *atoms-derivs-subset*: *atoms* (*derivs* *w* *r*)  $\subseteq$  *atoms* *r*

**by** (*induction* *w* *arbitrary*: *r*) (*auto* *dest*: *atoms-deriv-subset*[*THEN* *subsetD*])

**lemma** *lang-deriv*: *lang* (*deriv* *c* *r*) = *Deriv* *c* (*lang* *r*)

**by** (*induct* *r*) (*simp-all* *add*: *nullable-iff*)

**lemma** *lang-derivs*: *lang* (*derivs* *s* *r*) = *Derivs* *s* (*lang* *r*)

**by** (*induct* *s* *arbitrary*: *r*) (*simp-all* *add*: *lang-deriv*)

A regular expression matcher:

**definition** *matcher* :: 'a *rexp*  $\Rightarrow$  'a *list*  $\Rightarrow$  *bool* **where**

*matcher* *r* *s* = *nullable* (*derivs* *s* *r*)

**lemma** *matcher-correctness*: *matcher* *r* *s*  $\longleftrightarrow$  *s*  $\in$  *lang* *r*

**by** (*induct* *s* *arbitrary*: *r*)

(*simp-all* *add*: *nullable-iff* *lang-deriv* *matcher-def* *Deriv-def*)

## 8.2 Antimirov's partial derivatives

**abbreviation**

*Times* *rs* *r*  $\equiv$  ( $\bigcup$  *r'*  $\in$  *rs*. {*Times* *r'* *r*})

**lemma** *Times-eq-image*:  
 $Times\ rs\ r = (\lambda r'. Times\ r'\ r) \text{ ' } rs$   
**by** *auto*

**primrec**

$pderiv :: 'a \Rightarrow 'a\ rexp \Rightarrow 'a\ rexp\ set$

**where**

$pderiv\ c\ Zero = \{\}$   
 $| pderiv\ c\ One = \{\}$   
 $| pderiv\ c\ (Atom\ c') = (if\ c = c'\ then\ \{One\}\ else\ \{\})$   
 $| pderiv\ c\ (Plus\ r1\ r2) = (pderiv\ c\ r1) \cup (pderiv\ c\ r2)$   
 $| pderiv\ c\ (Times\ r1\ r2) =$   
 $\quad (if\ nullable\ r1\ then\ Times\ (pderiv\ c\ r1)\ r2 \cup pderiv\ c\ r2\ else\ Times\ (pderiv$   
 $\quad c\ r1)\ r2)$   
 $| pderiv\ c\ (Star\ r) = Times\ (pderiv\ c\ r)\ (Star\ r)$

**primrec**

$pderivs :: 'a\ list \Rightarrow 'a\ rexp \Rightarrow ('a\ rexp)\ set$

**where**

$pderivs\ []\ r = \{r\}$   
 $| pderivs\ (c\ \#s)\ r = \bigcup (pderivs\ s\ \text{' } pderiv\ c\ r)$

**abbreviation**

$pderiv\text{-}set :: 'a \Rightarrow 'a\ rexp\ set \Rightarrow 'a\ rexp\ set$

**where**

$pderiv\text{-}set\ c\ rs \equiv \bigcup (pderiv\ c\ \text{' } rs)$

**abbreviation**

$pderivs\text{-}set :: 'a\ list \Rightarrow 'a\ rexp\ set \Rightarrow 'a\ rexp\ set$

**where**

$pderivs\text{-}set\ s\ rs \equiv \bigcup (pderivs\ s\ \text{' } rs)$

**lemma** *pderivs-append*:

$pderivs\ (s1\ @\ s2)\ r = \bigcup (pderivs\ s2\ \text{' } pderivs\ s1\ r)$

**by** (*induct s1 arbitrary: r*) (*simp-all*)

**lemma** *pderivs-snoc*:

**shows**  $pderivs\ (s\ @\ [c])\ r = pderiv\text{-}set\ c\ (pderivs\ s\ r)$

**by** (*simp add: pderivs-append*)

**lemma** *pderivs-simps* [*simp*]:

**shows**  $pderivs\ s\ Zero = (if\ s = []\ then\ \{Zero\}\ else\ \{\})$

**and**  $pderivs\ s\ One = (if\ s = []\ then\ \{One\}\ else\ \{\})$

**and**  $pderivs\ s\ (Plus\ r1\ r2) = (if\ s = []\ then\ \{Plus\ r1\ r2\}\ else\ (pderivs\ s\ r1) \cup (pderivs\ s\ r2))$

**by** (*induct s*) (*simp-all*)

**lemma** *pderivs-Atom*:

**shows**  $pderiv\ s\ (Atom\ c) \subseteq \{Atom\ c,\ One\}$   
**by**  $(induct\ s)\ (simp\ all)$

### 8.3 Relating left-quotients and partial derivatives

**lemma** *Deriv-pderiv*:

**shows**  $Deriv\ c\ (lang\ r) = \bigcup (lang\ 'pderiv\ c\ r)$   
**by**  $(induct\ r)\ (auto\ simp\ add:\ nullable\ iff\ conc\ UNION\ distrib)$

**lemma** *Derivs-pderivs*:

**shows**  $Derivs\ s\ (lang\ r) = \bigcup (lang\ 'pderivs\ s\ r)$   
**proof**  $(induct\ s\ arbitrary:\ r)$   
**case**  $(Cons\ c\ s)$   
**have**  $ih:\ \bigwedge r.\ Derivs\ s\ (lang\ r) = \bigcup (lang\ 'pderivs\ s\ r)$  **by** *fact*  
**have**  $Derivs\ (c\ \# \ s)\ (lang\ r) = Derivs\ s\ (Deriv\ c\ (lang\ r))$  **by** *simp*  
**also have**  $\dots = Derivs\ s\ (\bigcup (lang\ 'pderiv\ c\ r))$  **by**  $(simp\ add:\ Deriv\ pderiv)$   
**also have**  $\dots = Derivs\ s\ (lang\ '(pderiv\ c\ r))$   
**by**  $(auto\ simp\ add:\ Derivs\ def)$   
**also have**  $\dots = \bigcup (lang\ '(pderivs\ set\ s\ (pderiv\ c\ r)))$   
**using** *ih* **by** *auto*  
**also have**  $\dots = \bigcup (lang\ '(pderivs\ (c\ \# \ s)\ r))$  **by** *simp*  
**finally show**  $Derivs\ (c\ \# \ s)\ (lang\ r) = \bigcup (lang\ 'pderivs\ (c\ \# \ s)\ r)$ .  
**qed**  $(simp\ add:\ Derivs\ def)$

### 8.4 Relating derivatives and partial derivatives

**lemma** *deriv-pderiv*:

**shows**  $\bigcup (lang\ '(pderiv\ c\ r)) = lang\ (deriv\ c\ r)$   
**unfolding** *lang-deriv Deriv-pderiv* **by** *simp*

**lemma** *derivs-pderivs*:

**shows**  $\bigcup (lang\ '(pderivs\ s\ r)) = lang\ (derivs\ s\ r)$   
**unfolding** *lang-derivs Derivs-pderivs* **by** *simp*

### 8.5 Finiteness property of partial derivatives

**definition**

*pderivs-lang*  $:: 'a\ lang \Rightarrow 'a\ rexp \Rightarrow 'a\ rexp\ set$

**where**

*pderivs-lang*  $A\ r \equiv \bigcup x \in A.\ pderivs\ x\ r$

**lemma** *pderivs-lang-subsetI*:

**assumes**  $\bigwedge s.\ s \in A \Longrightarrow pderivs\ s\ r \subseteq C$   
**shows**  $pderivs\ lang\ A\ r \subseteq C$   
**using** *assms* **unfolding** *pderivs-lang-def* **by**  $(rule\ UN\ least)$

**lemma** *pderivs-lang-union*:

**shows**  $pderivs\ lang\ (A \cup B)\ r = (pderivs\ lang\ A\ r \cup pderivs\ lang\ B\ r)$   
**by**  $(simp\ add:\ pderivs\ lang\ def)$

**lemma** *pderivs-lang-subset*:  
**shows**  $A \subseteq B \implies \text{pderivs-lang } A \ r \subseteq \text{pderivs-lang } B \ r$   
**by** (*auto simp add: pderivs-lang-def*)

**definition**  
 $UNIV1 \equiv UNIV - \{\}\}$

**lemma** *pderivs-lang-Zero* [*simp*]:  
**shows**  $\text{pderivs-lang } UNIV1 \ Zero = \{\}$   
**unfolding** *UNIV1-def pderivs-lang-def* **by** *auto*

**lemma** *pderivs-lang-One* [*simp*]:  
**shows**  $\text{pderivs-lang } UNIV1 \ One = \{\}$   
**unfolding** *UNIV1-def pderivs-lang-def* **by** (*auto split: if-splits*)

**lemma** *pderivs-lang-Atom* [*simp*]:  
**shows**  $\text{pderivs-lang } UNIV1 \ (\text{Atom } c) = \{\text{One}\}$   
**unfolding** *UNIV1-def pderivs-lang-def*  
**apply** (*auto*)  
**apply** (*frule rev-subsetD*)  
**apply** (*rule pderivs-Atom*)  
**apply** (*simp*)  
**apply** (*case-tac xa*)  
**apply** (*auto split: if-splits*)  
**done**

**lemma** *pderivs-lang-Plus* [*simp*]:  
**shows**  $\text{pderivs-lang } UNIV1 \ (\text{Plus } r1 \ r2) = \text{pderivs-lang } UNIV1 \ r1 \cup \text{pderivs-lang } UNIV1 \ r2$   
**unfolding** *UNIV1-def pderivs-lang-def* **by** *auto*

Non-empty suffixes of a string (needed for the cases of *Times* and *Star* below)

**definition**  
 $PSuf \ s \equiv \{v. v \neq \{\} \wedge (\exists u. u @ v = s)\}$

**lemma** *PSuf-snoc*:  
**shows**  $PSuf \ (s @ [c]) = (PSuf \ s) @ @ \ {[c]} \cup \ {[c]}$   
**unfolding** *PSuf-def conc-def*  
**by** (*auto simp add: append-eq-append-conv2 append-eq-Cons-conv*)

**lemma** *PSuf-Union*:  
**shows**  $(\bigcup v \in PSuf \ s @ @ \ {[c]}. f \ v) = (\bigcup v \in PSuf \ s. f \ (v @ [c]))$   
**by** (*auto simp add: conc-def*)

**lemma** *pderivs-lang-snoc*:  
**shows**  $\text{pderivs-lang } (PSuf \ s @ @ \ {[c]}) \ r = (\text{pderiv-set } c \ (\text{pderivs-lang } (PSuf \ s) \ r))$   
**unfolding** *pderivs-lang-def*

by (simp add: PSuf-Union pderivs-snoc)

**lemma** *pderivs-Times*:

shows  $pderivs\ s\ (Times\ r1\ r2) \subseteq Times\ (pderivs\ s\ r1)\ r2 \cup (pderivs\text{-}lang\ (PSuf\ s)\ r2)$

**proof** (induct s rule: rev-induct)

case (snoc c s)

have ih:  $pderivs\ s\ (Times\ r1\ r2) \subseteq Times\ (pderivs\ s\ r1)\ r2 \cup (pderivs\text{-}lang\ (PSuf\ s)\ r2)$

by fact

have  $pderivs\ (s\ @\ [c])\ (Times\ r1\ r2) = pderiv\text{-}set\ c\ (pderivs\ s\ (Times\ r1\ r2))$

by (simp add: pderivs-snoc)

also have  $\dots \subseteq pderiv\text{-}set\ c\ (Times\ (pderivs\ s\ r1)\ r2 \cup (pderivs\text{-}lang\ (PSuf\ s)\ r2))$

using ih by fastforce

also have  $\dots = pderiv\text{-}set\ c\ (Times\ (pderivs\ s\ r1)\ r2) \cup pderiv\text{-}set\ c\ (pderivs\text{-}lang\ (PSuf\ s)\ r2)$

by (simp)

also have  $\dots = pderiv\text{-}set\ c\ (Times\ (pderivs\ s\ r1)\ r2) \cup pderivs\text{-}lang\ (PSuf\ s\ @@\ \{[c]\})\ r2$

by (simp add: pderivs-lang-snoc)

also

have  $\dots \subseteq pderiv\text{-}set\ c\ (Times\ (pderivs\ s\ r1)\ r2) \cup pderiv\ c\ r2 \cup pderivs\text{-}lang\ (PSuf\ s\ @@\ \{[c]\})\ r2$

by auto

also

have  $\dots \subseteq Times\ (pderiv\text{-}set\ c\ (pderivs\ s\ r1))\ r2 \cup pderiv\ c\ r2 \cup pderivs\text{-}lang\ (PSuf\ s\ @@\ \{[c]\})\ r2$

by (auto simp add: if-splits)

also have  $\dots = Times\ (pderivs\ (s\ @\ [c])\ r1)\ r2 \cup pderiv\ c\ r2 \cup pderivs\text{-}lang\ (PSuf\ s\ @@\ \{[c]\})\ r2$

by (simp add: pderivs-snoc)

also have  $\dots \subseteq Times\ (pderivs\ (s\ @\ [c])\ r1)\ r2 \cup pderivs\text{-}lang\ (PSuf\ (s\ @\ [c]))\ r2$

unfolding pderivs-lang-def by (auto simp add: PSuf-snoc)

finally show ?case .

qed (simp)

**lemma** *pderivs-lang-Times-aux1*:

assumes a:  $s \in UNIV1$

shows  $pderivs\text{-}lang\ (PSuf\ s)\ r \subseteq pderivs\text{-}lang\ UNIV1\ r$

using a unfolding UNIV1-def PSuf-def pderivs-lang-def by auto

**lemma** *pderivs-lang-Times-aux2*:

assumes a:  $s \in UNIV1$

shows  $Times\ (pderivs\ s\ r1)\ r2 \subseteq Times\ (pderivs\text{-}lang\ UNIV1\ r1)\ r2$

using a unfolding pderivs-lang-def by auto

**lemma** *pderivs-lang-Times*:

```

shows pderivs-lang UNIV1 (Times r1 r2) ⊆ Timess (pderivs-lang UNIV1 r1) r2
  ∪ pderivs-lang UNIV1 r2
apply(rule pderivs-lang-subsetI)
apply(rule subset-trans)
apply(rule pderivs-Times)
using pderivs-lang-Times-aux1 pderivs-lang-Times-aux2
apply auto
apply blast
done

```

**lemma** *pderivs-Star*:

```

assumes a: s ≠ []
shows pderivs s (Star r) ⊆ Timess (pderivs-lang (PSuf s) r) (Star r)
using a
proof (induct s rule: rev-induct)
  case (snoc c s)
    have ih: s ≠ [] ⇒ pderivs s (Star r) ⊆ Timess (pderivs-lang (PSuf s) r) (Star r)
    by fact
    { assume asm: s ≠ []
      have pderivs (s @ [c]) (Star r) = pderiv-set c (pderivs s (Star r)) by (simp add: pderivs-snoc)
      also have ... ⊆ pderiv-set c (Timess (pderivs-lang (PSuf s) r) (Star r))
        using ih[OF asm] by fast
      also have ... ⊆ Timess (pderiv-set c (pderivs-lang (PSuf s) r)) (Star r) ∪ pderiv c (Star r)
        by (auto split: if-splits)
      also have ... ⊆ Timess (pderivs-lang (PSuf (s @ [c])) r) (Star r) ∪ (Timess (pderiv c r) (Star r))
        by (simp only: PSuf-snoc pderivs-lang-snoc pderivs-lang-union)
        (auto simp add: pderivs-lang-def)
      also have ... = Timess (pderivs-lang (PSuf (s @ [c])) r) (Star r)
        by (auto simp add: PSuf-snoc PSuf-Union pderivs-snoc pderivs-lang-def)
      finally have ?case .
    }
  moreover
  { assume asm: s = []
    then have ?case by (auto simp add: pderivs-lang-def pderivs-snoc PSuf-def)
  }
  ultimately show ?case by blast
qed (simp)

```

**lemma** *pderivs-lang-Star*:

```

shows pderivs-lang UNIV1 (Star r) ⊆ Timess (pderivs-lang UNIV1 r) (Star r)
apply(rule pderivs-lang-subsetI)
apply(rule subset-trans)
apply(rule pderivs-Star)
apply(simp add: UNIV1-def)
apply(simp add: UNIV1-def PSuf-def)
apply(auto simp add: pderivs-lang-def)

```

**done**

**lemma** *finite-Times* [*simp*]:  
  **assumes** *a*: *finite A*  
  **shows** *finite (Times A r)*  
**using** *a* **by** *auto*

**lemma** *finite-pderivs-lang-UNIV1*:  
  **shows** *finite (pderivs-lang UNIV1 r)*  
**apply**(*induct r*)  
**apply**(*simp-all add:*  
  *finite-subset[OF pderivs-lang-Times]*  
  *finite-subset[OF pderivs-lang-Star]*)  
**done**

**lemma** *pderivs-lang-UNIV*:  
  **shows** *pderivs-lang UNIV r = pderivs [] r ∪ pderivs-lang UNIV1 r*  
**unfolding** *UNIV1-def pderivs-lang-def*  
**by** *blast*

**lemma** *finite-pderivs-lang-UNIV*:  
  **shows** *finite (pderivs-lang UNIV r)*  
**unfolding** *pderivs-lang-UNIV*  
**by** (*simp add: finite-pderivs-lang-UNIV1*)

**lemma** *finite-pderivs-lang*:  
  **shows** *finite (pderivs-lang A r)*  
**by** (*metis finite-pderivs-lang-UNIV pderivs-lang-subset rev-finite-subset subset-UNIV*)

The following relationship between the alphabetic width of regular expressions (called *awidth* below) and the number of partial derivatives was proved by Antimirov [1] and formalized by Max Haslbeck.

**fun** *awidth* :: '*a rexp* ⇒ *nat* **where**  
*awidth Zero = 0* |  
*awidth One = 0* |  
*awidth (Atom a) = 1* |  
*awidth (Plus r1 r2) = awidth r1 + awidth r2* |  
*awidth (Times r1 r2) = awidth r1 + awidth r2* |  
*awidth (Star r1) = awidth r1*

**lemma** *card-Times-pderivs-lang-le*:  
  *card (Times (pderivs-lang A r) s) ≤ card (pderivs-lang A r)*  
  **using** *finite-pderivs-lang unfolding Times-eq-image* **by** (*rule card-image-le*)

**lemma** *card-pderivs-lang-UNIV1-le-awidth*: *card (pderivs-lang UNIV1 r) ≤ awidth r*  
**proof** (*induction r*)  
  **case** (*Plus r1 r2*)  
  **have** *card (pderivs-lang UNIV1 (Plus r1 r2)) = card (pderivs-lang UNIV1 r1 ∪*

```

pderivs-lang UNIV1 r2) by simp
  also have ...  $\leq$  card (pderivs-lang UNIV1 r1) + card (pderivs-lang UNIV1 r2)
    by(simp add: card-Un-le)
  also have ...  $\leq$  awidth (Plus r1 r2) using Plus.IH by simp
  finally show ?case .
next
  case (Times r1 r2)
    have card (pderivs-lang UNIV1 (Times r1 r2))  $\leq$  card (Times (pderivs-lang UNIV1 r1) r2  $\cup$  pderivs-lang UNIV1 r2)
      by (simp add: card-mono finite-pderivs-lang pderivs-lang-Times)
    also have ...  $\leq$  card (Times (pderivs-lang UNIV1 r1) r2) + card (pderivs-lang UNIV1 r2)
      by (simp add: card-Un-le)
    also have ...  $\leq$  card (pderivs-lang UNIV1 r1) + card (pderivs-lang UNIV1 r2)
      by (simp add: card-Times-pderivs-lang-le)
    also have ...  $\leq$  awidth (Times r1 r2) using Times.IH by simp
    finally show ?case .
next
  case (Star r)
    have card (pderivs-lang UNIV1 (Star r))  $\leq$  card (Times (pderivs-lang UNIV1 r) (Star r))
      by (simp add: card-mono finite-pderivs-lang pderivs-lang-Star)
    also have ...  $\leq$  card (pderivs-lang UNIV1 r) by (rule card-Times-pderivs-lang-le)
    also have ...  $\leq$  awidth (Star r) by (simp add: Star.IH)
    finally show ?case .
qed (auto)

```

Antimirov's Theorem 3.4:

```

theorem card-pderivs-lang-UNIV-le-awidth: card (pderivs-lang UNIV r)  $\leq$  awidth r + 1
proof –
  have card (insert r (pderivs-lang UNIV1 r))  $\leq$  Suc (card (pderivs-lang UNIV1 r))
    by(auto simp: card-insert-if[OF finite-pderivs-lang-UNIV1])
  also have ...  $\leq$  Suc (awidth r) by(simp add: card-pderivs-lang-UNIV1-le-awidth)
  finally show ?thesis by(simp add: pderivs-lang-UNIV)
qed

```

Antimirov's Corollary 3.5:

```

corollary card-pderivs-lang-le-awidth: card (pderivs-lang A r)  $\leq$  awidth r + 1
by(rule order-trans[OF card-mono[OF finite-pderivs-lang-UNIV pderivs-lang-subset[OF subset-UNIV]] card-pderivs-lang-UNIV-le-awidth])

```

**end**

## 9 Deciding Regular Expression Equivalence (2)

**theory** *pEquivalence-Checking*



**imports** *Equivalence-Checking Derivatives*  
**begin**

Similar to theory *Regular-Sets.Equivalence-Checking*, but with partial derivatives instead of derivatives.

Lifting many notions to sets:

**definition**  $Lang\ R == UN\ r:R.\ lang\ r$

**definition**  $Nullable\ R == EX\ r:R.\ nullable\ r$

**definition**  $Pderiv\ a\ R == UN\ r:R.\ pderiv\ a\ r$

**definition**  $Atoms\ R == UN\ r:R.\ atoms\ r$

**lemma** *Atoms-pderiv*:  $Atoms(pderiv\ a\ r) \subseteq atoms\ r$

**apply** (*induct*  $r$ )

**apply** (*auto simp*: *Atoms-def UN-subset-iff*)

**apply** (*fastforce*)+

**done**

**lemma** *Atoms-Pderiv*:  $Atoms(Pderiv\ a\ R) \subseteq Atoms\ R$

**using** *Atoms-pderiv* **by** (*fastforce simp*: *Atoms-def Pderiv-def*)

**lemma** *pderiv-no-occurrence*:

$x \notin atoms\ r \implies pderiv\ x\ r = \{\}$

**by** (*induct*  $r$ ) *auto*

**lemma** *Pderiv-no-occurrence*:

$x \notin Atoms\ R \implies Pderiv\ x\ R = \{\}$

**by**(*auto simp*:*pderiv-no-occurrence Atoms-def Pderiv-def*)

**lemma** *Deriv-Lang*:  $Deriv\ c\ (Lang\ R) = Lang\ (Pderiv\ c\ R)$

**by**(*auto simp*: *Deriv-pderiv Pderiv-def Lang-def*)

**lemma** *Nullable-pderiv[simp]*:  $Nullable(pderivs\ w\ r) = (w : lang\ r)$

**apply**(*induction*  $w$  *arbitrary*:  $r$ )

**apply** (*simp add*: *Nullable-def nullable-iff singleton-iff*)

**using** *eqset-imp-iff*[*OF Deriv-pderiv*[**where**  $'a = 'a$ ]]

**apply** (*simp add*: *Nullable-def Deriv-def*)

**done**

**type-synonym**  $'a\ Rexp\ pair = 'a\ rexp\ set * 'a\ rexp\ set$

**type-synonym**  $'a\ Rexp\ pairs = 'a\ Rexp\ pair\ list$

**definition** *is-Bisimulation*  $:: 'a\ list \Rightarrow 'a\ Rexp\ pairs \Rightarrow bool$

**where**

*is-Bisimulation*  $as\ ps =$

$(\forall (R,S) \in set\ ps.\ Atoms\ R \cup Atoms\ S \subseteq set\ as \wedge$

$(Nullable\ R \longleftrightarrow Nullable\ S) \wedge$

$(\forall a \in \text{set } as. (Pderiv\ a\ R, Pderiv\ a\ S) \in \text{set } ps))$

**lemma** *Bisim-Lang-eq*:  
**assumes** *Bisim*: *is-Bisimulation as ps*  
**assumes**  $(R, S) \in \text{set } ps$   
**shows**  $\text{Lang } R = \text{Lang } S$   
**proof** –  
**define** *ps'* **where**  $ps' = (\{\}, \{\}) \# ps$   
**from** *Bisim* **have** *Bisim'*: *is-Bisimulation as ps'*  
**by** (*fastforce simp: ps'-def is-Bisimulation-def UN-subset-iff Pderiv-def Atoms-def*)  
**let**  $?R = \lambda K L. (\exists (R,S) \in \text{set } ps'. K = \text{Lang } R \wedge L = \text{Lang } S)$   
**show** *?thesis*  
**proof** (*rule language-coinduct[where R=?R]*)  
**from**  $(R,S) \in \text{set } ps$   
**have**  $(R,S) \in \text{set } ps'$  **by** (*auto simp: ps'-def*)  
**thus**  $?R (\text{Lang } R) (\text{Lang } S)$  **by** *auto*  
**next**  
**fix** *K L* **assume**  $?R\ K\ L$   
**then obtain** *R S* **where**  $rs: (R, S) \in \text{set } ps'$   
**and** *KL*:  $K = \text{Lang } R\ L = \text{Lang } S$  **by** *auto*  
**with** *Bisim'* **have**  $\text{Nullable } R \longleftrightarrow \text{Nullable } S$   
**by** (*auto simp: is-Bisimulation-def*)  
**thus**  $\square \in K \longleftrightarrow \square \in L$   
**by** (*auto simp: nullable-iff KL Nullable-def Lang-def*)  
**fix** *a*  
**show**  $?R (\text{Deriv } a\ K) (\text{Deriv } a\ L)$   
**proof** *cases*  
**assume**  $a \in \text{set } as$   
**with** *rs Bisim'*  
**have**  $(Pderiv\ a\ R, Pderiv\ a\ S) \in \text{set } ps'$   
**by** (*auto simp: is-Bisimulation-def*)  
**thus** *?thesis* **by** (*fastforce simp: KL Deriv-Lang*)  
**next**  
**assume**  $a \notin \text{set } as$   
**with** *Bisim' rs*  
**have**  $a \notin \text{Atoms } R \cup \text{Atoms } S$   
**by** (*fastforce simp: is-Bisimulation-def UN-subset-iff*)  
**then have**  $Pderiv\ a\ R = \{\}\ Pderiv\ a\ S = \{\}$   
**by** (*metis Pderiv-no-occurrence Un-iff*)  
**then have**  $\text{Deriv } a\ K = \text{Lang } \{\}\ \text{Deriv } a\ L = \text{Lang } \{\}$   
**unfolding** *KL Deriv-Lang* **by** *auto*  
**thus** *?thesis* **by** (*auto simp: ps'-def*)  
**qed**  
**qed**  
**qed**

## 9.1 Closure computation

**fun** *test* ::  $'a\ \text{Rexp-pairs} * 'a\ \text{Rexp-pairs} \Rightarrow \text{bool}$  **where**

$test (ws, ps) = (case\ ws\ of\ [] \Rightarrow False \mid (R,S)\#- \Rightarrow Nullable\ R = Nullable\ S)$

```
fun step :: 'a list  $\Rightarrow$ 
  'a Rexp-pairs * 'a Rexp-pairs  $\Rightarrow$  'a Rexp-pairs * 'a Rexp-pairs
where step as (ws,ps) =
  (let
    (R,S) = hd ws;
    ps' = (R,S) # ps;
    succs = map ( $\lambda a.$  (Pderiv a R, Pderiv a S)) as;
    new = filter ( $\lambda p.$   $p \notin set\ ps \cup set\ ws$ ) succs
  in (remdups new @ tl ws, ps')
```

**definition** closure ::  
 'a list  $\Rightarrow$  'a Rexp-pairs \* 'a Rexp-pairs  
 $\Rightarrow$  ('a Rexp-pairs \* 'a Rexp-pairs) option **where**  
 closure as = while-option test (step as)

**definition** pre-Bisim :: 'a list  $\Rightarrow$  'a rexp set  $\Rightarrow$  'a rexp set  $\Rightarrow$   
 'a Rexp-pairs \* 'a Rexp-pairs  $\Rightarrow$  bool  
**where**  
 pre-Bisim as R S = ( $\lambda(ws,ps).$   
 ((R,S)  $\in set\ ws \cup set\ ps$ )  $\wedge$   
 ( $\forall (R,S) \in set\ ws \cup set\ ps.$  Atoms R  $\cup$  Atoms S  $\subseteq set\ as$ )  $\wedge$   
 ( $\forall (R,S) \in set\ ps.$  (Nullable R  $\longleftrightarrow$  Nullable S)  $\wedge$   
 ( $\forall a \in set\ as.$  (Pderiv a R, Pderiv a S)  $\in set\ ps \cup set\ ws$ )))

**lemma** step-set-eq:  $\llbracket test (ws,ps); step\ as\ (ws,ps) = (ws',ps') \rrbracket$   
 $\implies set\ ws' \cup set\ ps' =$   
 $set\ ws \cup set\ ps$   
 $\cup (\bigcup a \in set\ as. \{(Pderiv\ a\ (fst(hd\ ws)), Pderiv\ a\ (snd(hd\ ws)))\})$   
**by**(auto split: list.splits)

**theorem** closure-sound:

**assumes** result: closure as ( $[(R,S)], []$ ) = Some( $[], ps$ )

**and** atoms: Atoms R  $\cup$  Atoms S  $\subseteq set\ as$

**shows** Lang R = Lang S

**proof**–

{ **fix** st

**have** pre-Bisim as R S st  $\implies test\ st \implies pre-Bisim\ as\ R\ S$  (step as st)

**unfolding** pre-Bisim-def

**proof**(split prod.splits, elim case-prodE conjE, intro allI impI conjI, goal-cases)

**case** 1 **thus** ?case **by**(auto split: list.splits)

**next**

**case** prems: ( $2\ ws\ ps\ ws'\ ps'$ )

**note** prems(2)[simp]

**from**  $\langle test\ st \rangle$  **obtain** wstl R S **where** [simp]:  $ws = (R,S)\#wstl$

**by** (auto split: list.splits)

**from**  $\langle step\ as\ st = (ws',ps') \rangle$  **obtain** P **where** [simp]:  $ps' = (R,S) \# ps$

**and** [simp]:  $ws' = remdups(filter\ P\ (map\ (\lambda a.$  (Pderiv a R, Pderiv a S)))

```

as)) @ wstl
  by auto
  have  $\forall (R', S') \in \text{set wstl} \cup \text{set ps}'. \text{Atoms } R' \cup \text{Atoms } S' \subseteq \text{set as}$ 
    using prems(4) by auto
  moreover
  have  $\forall a \in \text{set as}. \text{Atoms}(Pderiv a R) \cup \text{Atoms}(Pderiv a S) \subseteq \text{set as}$ 
    using prems(4) by simp (metis (lifting) Atoms-Pderiv order-trans)
  ultimately show ?case by simp blast
next
  case 3 thus ?case
    apply (clarsimp simp: image-iff split: prod.splits list.splits)
    by hypsubst-thin metis
qed
}
moreover
from atoms
have pre-Bisim as R S ([[R,S]], []) by (simp add: pre-Bisim-def)
ultimately have pre-Bisim-ps: pre-Bisim as R S ([], ps)
  by (rule while-option-rule[OF - result[unfolded closure-def]])
then have is-Bisimulation as ps (R,S)  $\in$  set ps
  by (auto simp: pre-Bisim-def is-Bisimulation-def)
thus Lang R = Lang S by (rule Bisim-Lang-eq)
qed

```

## 9.2 The overall procedure

**definition** *check-equiv* :: 'a rexp  $\Rightarrow$  'a rexp  $\Rightarrow$  bool

**where**

```

check-equiv r s =
  (case closure (add-atoms r (add-atoms s [])) ([[r], {s}], []) of
    Some([],-)  $\Rightarrow$  True | -  $\Rightarrow$  False)

```

**lemma** *soundness*: **assumes** *check-equiv r s* **shows** *lang r = lang s*

**proof** –

```

let ?as = add-atoms r (add-atoms s [])
obtain ps where 1: closure ?as ([[r], {s}], []) = Some([], ps)
  using assms by (auto simp: check-equiv-def split: option.splits list.splits)
then have lang r = lang s
  by(rule closure-sound[of - {r} {s}, simplified Lang-def, simplified])
  (auto simp: set-add-atoms Atoms-def)
thus lang r = lang s by simp
qed

```

Test:

**lemma** *check-equiv*

```

(Plus One (Times (Atom 0) (Star(Atom 0))))
(Star(Atom(0::nat)))

```

**by** *eval*

### 9.3 Termination and Completeness

**definition**  $PDERIVS :: 'a \text{ rexp set} \Rightarrow 'a \text{ rexp set}$  **where**  
 $PDERIVS R = (UN r:R. pderivs-lang UNIV r)$

**lemma**  $PDERIVS-incr[simp]: R \subseteq PDERIVS R$   
**apply**( $auto \ simp \ add: PDERIVS-def \ pderivs-lang-def$ )  
**by** ( $metis \ pderivs.simps(1) \ insertI1$ )

**lemma**  $Pderiv-PDERIVS: \text{assumes } R' \subseteq PDERIVS R \text{ shows } Pderiv \ a \ R' \subseteq PDERIVS R$

**proof**

**fix**  $r$  **assume**  $r : Pderiv \ a \ R'$   
**then obtain**  $r' \text{ where } r' : R' \ r : pderiv \ a \ r'$  **by**( $auto \ simp: Pderiv-def$ )  
**from**  $\langle r' : R' \rangle \langle R' \subseteq PDERIVS R \rangle$  **obtain**  $s \ w$  **where**  $s : R \ r' : pderivs \ w \ s$   
**by**( $auto \ simp: PDERIVS-def \ pderivs-lang-def$ )  
**hence**  $r \in pderivs \ (w \ @ \ [a]) \ s$  **using**  $\langle r : pderiv \ a \ r' \rangle$  **by**( $auto \ simp \ add: pderivs-snoc$ )  
**thus**  $r : PDERIVS R$  **using**  $\langle s : R \rangle$  **by**( $auto \ simp: PDERIVS-def \ pderivs-lang-def$ )  
**qed**

**lemma**  $finite-PDERIVS: finite \ R \Longrightarrow finite(PDERIVS R)$   
**by**( $simp \ add: PDERIVS-def \ finite-pderivs-lang-UNIV$ )

**lemma**  $closure-Some: \text{assumes } finite \ R0 \ finite \ S0 \text{ shows } \exists p. \text{closure as } ([[(R0, S0)], []]) = Some \ p$

**proof**( $unfold \ closure-def$ )

**let**  $?Inv = \% (ws, bs). \text{distinct } ws \wedge (ALL (R, S) : \text{set } ws. R \subseteq PDERIVS R0 \wedge S \subseteq PDERIVS S0 \wedge (R, S) \notin \text{set } bs)$

**let**  $?m1 = \% bs. Pow(PDERIVS R0) \times Pow(PDERIVS S0) - \text{set } bs$

**let**  $?m2 = \% (ws, bs). \text{card} (?m1 \ bs)$

**have**  $Inv0: ?Inv \ ([[(R0, S0)], []])$  **by**  $simp$

**{ fix**  $s$  **assume**  $test \ s \ ?Inv \ s$

**obtain**  $ws \ bs$  **where**  $[simp]: s = (ws, bs)$  **by**  $fastforce$

**from**  $\langle test \ s \rangle$  **obtain**  $R \ S \ ws'$  **where**  $[simp]: ws = (R, S) \# ws'$

**by**( $auto \ split: prod.splits \ list.splits$ )

**let**  $?bs' = (R, S) \# bs$

**let**  $?succs = \text{map } (\lambda a. (Pderiv \ a \ R, Pderiv \ a \ S)) \ as$

**let**  $?new = \text{filter } (\lambda p. p \notin \text{set } bs \cup \text{set } ws) \ ?succs$

**let**  $?ws' = \text{remdups } ?new \ @ \ ws'$

**have**  $*$ :  $?Inv \ (\text{step } as \ s)$

**proof**  $-$

**from**  $\langle ?Inv \ s \rangle$  **have**  $\text{distinct } ?ws'$  **by**  $auto$

**have**  $ALL (R, S) : \text{set } ?ws'. R \subseteq PDERIVS R0 \wedge S \subseteq PDERIVS S0 \wedge (R, S)$

$\notin \text{set } ?bs'$  **using**  $\langle ?Inv \ s \rangle$

**by**( $simp \ add: Ball-def \ image-iff$ ) ( $metis \ Pderiv-PDERIVS$ )

**with**  $\langle \text{distinct } ?ws' \rangle$  **show**  $?thesis$  **by**( $simp$ )

**qed**

**have**  $?m2(\text{step } as \ s) < ?m2 \ s$

**proof**  $-$

**have**  $fin: finite(?m1 \ bs)$

```

    by(metis assms finite-Diff finite-PDERIVS finite-cartesian-product finite-Pow-iff)
    have  $?m2(\text{step } as \ s) < ?m2 \ s$  using  $\langle ?Inv \ s \rangle$  psubset-card-mono[OF  $\langle$ finite( $?m1 \ bs$ ) $\rangle$ ]
    apply (simp split: prod.split-asm)
    by (metis Diff-iff Pow-iff SigmaI fin card-gt-0-iff diff-Suc-less emptyE)
    then show  $?thesis$  using  $\langle ?Inv \ s \rangle$  by simp
  qed
  note * and this
} note step = this
show  $\exists p. \text{while-option test } (\text{step } as) \ ([(\{R0, S0\}], []) = \text{Some } p$ 
  by(rule measure-while-option-Some [where  $P = ?Inv$  and  $f = ?m2, OF - Inv0$ ])(simp add: step)
qed

```

**theorem** *closure-Some-Inv*: **assumes** *closure as*  $([\{\{r\}, \{s\}\}], []) = \text{Some } p$   
**shows**  $\forall (R, S) \in \text{set}(fst \ p). \exists w. R = pderiv \ w \ r \wedge S = pderiv \ w \ s$  (**is**  $?Inv \ p$ )  
**proof** –

```

from assms have 1: while-option test  $(\text{step } as) \ ([(\{r\}, \{s\}\)], []) = \text{Some } p$ 
  by(simp add: closure-def)
have Inv0: ?Inv  $([\{\{r\}, \{s\}\}], [])$  by simp (metis pderivs.simps(1))
{ fix p assume  $?Inv \ p \ \text{test } p$ 
  obtain ws bs where [simp]:  $p = (ws, bs)$  by fastforce
  from  $\langle \text{test } p \rangle$  obtain R S ws' where [simp]:  $ws = (R, S) \# ws'$ 
    by(auto split: prod.splits list.splits)
  let  $?succs = \text{map } (\lambda a. (Pderiv \ a \ R, Pderiv \ a \ S)) \ as$ 
  let  $?new = \text{filter } (\lambda p. p \notin \text{set } bs \cup \text{set } ws) \ ?succs$ 
  let  $?ws' = \text{remdups } ?new \ @ \ ws'$ 
  from  $\langle ?Inv \ p \rangle$  obtain w where [simp]:  $R = pderiv \ w \ r \ S = pderiv \ w \ s$ 
    by auto
  { fix x assume  $x : \text{set } as$ 
    have EX w. Pderiv x R = pderiv w r  $\wedge Pderiv \ x \ S = pderiv \ w \ s$ 
      by(rule-tac x=w@[x] in exI)(simp add: pderivs-append Pderiv-def)
    }
  with  $\langle ?Inv \ p \rangle$  have  $?Inv \ (\text{step } as \ p)$  by auto
} note Inv-step = this
show  $?thesis$ 
  apply(rule while-option-rule[OF - 1])
  apply(erule (1) Inv-step)
  apply(rule Inv0)
  done
qed

```

**lemma** *closure-complete*: **assumes**  $\text{lang } r = \text{lang } s$   
**shows**  $\exists X \ bs. \text{closure as } ([(\{r\}, \{s\}\)], []) = \text{Some}([\ ], bs)$  (**is**  $?C$ )  
**proof**(*rule ccontr*)  
**assume**  $\neg ?C$   
**then obtain** *R S ws bs*  
**where** *cl: closure as*  $([\{\{r\}, \{s\}\}], []) = \text{Some}((R, S) \# ws, bs)$   
**using** *closure-Some*[*of*  $\{r\} \ \{s\}$ , *simplified*]

```

  by (metis (opaque-lifting, no-types) list.exhaust prod.exhaust)
  from assms closure-Some-Inv[OF this]
  while-option-stop[OF cl[unfolded closure-def]]
  show False by auto
qed

```

```

corollary completeness: lang r = lang s  $\implies$  check-equiv r s
by(auto simp add: check-equiv-def dest!: closure-complete
  split: option.split list.split)

```

**end**

## 10 Extended Regular Expressions

```

theory Regular-Exp2
imports Regular-Set
begin

```

```

datatype (atoms: 'a) rexp =
  is-Zero: Zero |
  is-One: One |
  Atom 'a |
  Plus ('a rexp) ('a rexp) |
  Times ('a rexp) ('a rexp) |
  Star ('a rexp) |
  Not ('a rexp) |
  Inter ('a rexp) ('a rexp)

```

```

context
fixes S :: 'a set
begin

```

```

primrec lang :: 'a rexp  $\implies$  'a lang where
  lang Zero = {} |
  lang One = {[]} |
  lang (Atom a) = {[a]} |
  lang (Plus r s) = (lang r) Un (lang s) |
  lang (Times r s) = conc (lang r) (lang s) |
  lang (Star r) = star(lang r) |
  lang (Not r) = lists S - lang r |
  lang (Inter r s) = (lang r Int lang s)

```

**end**

```

lemma lang-subset-lists: atoms r  $\subseteq$  S  $\implies$  lang S r  $\subseteq$  lists S
by(induction r)(auto simp: conc-subset-lists star-subset-lists)

```

```

primrec nullable :: 'a rexp  $\implies$  bool where
  nullable Zero = False |

```

```

nullable One = True |
nullable (Atom c) = False |
nullable (Plus r1 r2) = (nullable r1 ∨ nullable r2) |
nullable (Times r1 r2) = (nullable r1 ∧ nullable r2) |
nullable (Star r) = True |
nullable (Not r) = (¬ (nullable r)) |
nullable (Inter r s) = (nullable r ∧ nullable s)

```

**lemma** *nullable-iff*:  $\text{nullable } r \longleftrightarrow \square \in \text{lang } S \ r$   
**by** (*induct r*) (*auto simp add: conc-def split: if-splits*)

**end**

## 11 Deciding Equivalence of Extended Regular Expressions

**theory** *Equivalence-Checking2*  
**imports** *Regular-Exp2 HOL-Library.While-Combinator*  
**begin**

### 11.1 Term ordering

```

fun le-rexp :: nat rexp ⇒ nat rexp ⇒ bool
where
  le-rexp Zero - = True
| le-rexp - Zero = False
| le-rexp One - = True
| le-rexp - One = False
| le-rexp (Atom a) (Atom b) = (a ≤ b)
| le-rexp (Atom -) - = True
| le-rexp - (Atom -) = False
| le-rexp (Star r) (Star s) = le-rexp r s
| le-rexp (Star -) - = True
| le-rexp - (Star -) = False
| le-rexp (Not r) (Not s) = le-rexp r s
| le-rexp (Not -) - = True
| le-rexp - (Not -) = False
| le-rexp (Plus r r') (Plus s s') =
  (if r = s then le-rexp r' s' else le-rexp r s)
| le-rexp (Plus - -) - = True
| le-rexp - (Plus - -) = False
| le-rexp (Times r r') (Times s s') =
  (if r = s then le-rexp r' s' else le-rexp r s)
| le-rexp (Times - -) - = True
| le-rexp - (Times - -) = False
| le-rexp (Inter r r') (Inter s s') =
  (if r = s then le-rexp r' s' else le-rexp r s)

```



## 11.2 Normalizing operations

associativity, commutativity, idempotence, zero

**fun** *nPlus* :: *nat rexp*  $\Rightarrow$  *nat rexp*  $\Rightarrow$  *nat rexp*

**where**

```
nPlus Zero r = r
| nPlus r Zero = r
| nPlus (Plus r s) t = nPlus r (nPlus s t)
| nPlus r (Plus s t) =
  (if r = s then (Plus s t)
   else if le-rexp r s then Plus r (Plus s t)
   else Plus s (nPlus r t))
| nPlus r s =
  (if r = s then r
   else if le-rexp r s then Plus r s
   else Plus s r)
```

**lemma** *lang-nPlus[simp]*: *lang S* (*nPlus* *r* *s*) = *lang S* (*Plus* *r* *s*)

**by** (*induct* *r* *s* *rule*: *nPlus.induct*) *auto*

associativity, zero, one

**fun** *nTimes* :: *nat rexp*  $\Rightarrow$  *nat rexp*  $\Rightarrow$  *nat rexp*

**where**

```
nTimes Zero - = Zero
| nTimes - Zero = Zero
| nTimes One r = r
| nTimes r One = r
| nTimes (Times r s) t = Times r (nTimes s t)
| nTimes r s = Times r s
```

**lemma** *lang-nTimes[simp]*: *lang S* (*nTimes* *r* *s*) = *lang S* (*Times* *r* *s*)

**by** (*induct* *r* *s* *rule*: *nTimes.induct*) (*auto simp*: *conc-assoc*)

more optimisations:

**fun** *nInter* :: *nat rexp*  $\Rightarrow$  *nat rexp*  $\Rightarrow$  *nat rexp*

**where**

```
nInter Zero - = Zero
| nInter - Zero = Zero
| nInter r s = Inter r s
```

**lemma** *lang-nInter[simp]*: *lang S* (*nInter* *r* *s*) = *lang S* (*Inter* *r* *s*)

**by** (*induct* *r* *s* *rule*: *nInter.induct*) (*auto*)

**primrec** *norm* :: *nat rexp*  $\Rightarrow$  *nat rexp*

**where**

```
norm Zero = Zero
| norm One = One
| norm (Atom a) = Atom a
| norm (Plus r s) = nPlus (norm r) (norm s)
```

|  $norm (Times\ r\ s) = nTimes (norm\ r) (norm\ s)$   
|  $norm (Star\ r) = Star (norm\ r)$   
|  $norm (Not\ r) = Not (norm\ r)$   
|  $norm (Inter\ r1\ r2) = nInter (norm\ r1) (norm\ r2)$

**lemma** *lang-norm*[simp]:  $lang\ S (norm\ r) = lang\ S\ r$   
**by** (*induct r*) *auto*

### 11.3 Derivative

**primrec** *nderiv* ::  $nat \Rightarrow nat\ rexp \Rightarrow nat\ rexp$   
**where**

$nderiv - Zero = Zero$   
|  $nderiv - One = Zero$   
|  $nderiv\ a (Atom\ b) = (if\ a = b\ then\ One\ else\ Zero)$   
|  $nderiv\ a (Plus\ r\ s) = nPlus (nderiv\ a\ r) (nderiv\ a\ s)$   
|  $nderiv\ a (Times\ r\ s) =$   
 $(let\ r's = nTimes (nderiv\ a\ r)\ s$   
 $in\ if\ nullable\ r\ then\ nPlus\ r's (nderiv\ a\ s)\ else\ r's)$   
|  $nderiv\ a (Star\ r) = nTimes (nderiv\ a\ r) (Star\ r)$   
|  $nderiv\ a (Not\ r) = Not (nderiv\ a\ r)$   
|  $nderiv\ a (Inter\ r1\ r2) = nInter (nderiv\ a\ r1) (nderiv\ a\ r2)$

**lemma** *lang-nderiv*:  $a:S \Longrightarrow lang\ S (nderiv\ a\ r) = Deriv\ a (lang\ S\ r)$   
**by** (*induct r*) (*auto simp: Let-def nullable-iff*[**where**  $S=S$ ])

**lemma** *atoms-nPlus*[simp]:  $atoms (nPlus\ r\ s) = atoms\ r \cup atoms\ s$   
**by** (*induct r s rule: nPlus.induct*) *auto*

**lemma** *atoms-nTimes*:  $atoms (nTimes\ r\ s) \subseteq atoms\ r \cup atoms\ s$   
**by** (*induct r s rule: nTimes.induct*) *auto*

**lemma** *atoms-nInter*:  $atoms (nInter\ r\ s) \subseteq atoms\ r \cup atoms\ s$   
**by** (*induct r s rule: nInter.induct*) *auto*

**lemma** *atoms-norm*:  $atoms (norm\ r) \subseteq atoms\ r$   
**by** (*induct r*) (*auto dest!:subsetD[OF atoms-nTimes]subsetD[OF atoms-nInter]*)

**lemma** *atoms-nderiv*:  $atoms (nderiv\ a\ r) \subseteq atoms\ r$   
**by** (*induct r*) (*auto simp: Let-def dest!:subsetD[OF atoms-nTimes]subsetD[OF atoms-nInter]*)

### 11.4 Bisimulation between languages and regular expressions

**context**

**fixes**  $S :: 'a\ set$

**begin**

**coinductive** *bisimilar* ::  $'a\ lang \Rightarrow 'a\ lang \Rightarrow bool$  **where**

$K \subseteq lists\ S \Longrightarrow L \subseteq lists\ S$

$\Longrightarrow (\[] \in K \longleftrightarrow \[] \in L)$

$\implies (\bigwedge x. x:S \implies \text{bisimilar } (\text{Deriv } x \ K) \ (\text{Deriv } x \ L))$   
 $\implies \text{bisimilar } K \ L$

**lemma** *equal-if-bisimilar*:

**assumes**  $K \subseteq \text{lists } S$   $L \subseteq \text{lists } S$  *bisimilar*  $K \ L$  **shows**  $K = L$

**proof** (*rule set-eqI*)

**fix**  $w$

**from** *assms* **show**  $w \in K \longleftrightarrow w \in L$

**proof** (*induction w arbitrary: K L*)

**case** *Nil* **thus** *?case* **by** (*auto elim: bisimilar.cases*)

**next**

**case** (*Cons a w K L*)

**show** *?case*

**proof** *cases*

**assume**  $a : S$

**with**  $\langle \text{bisimilar } K \ L \rangle$  **have** *bisimilar* (*Deriv a K*) (*Deriv a L*)

**by** (*auto elim: bisimilar.cases*)

**then** **have**  $w \in \text{Deriv } a \ K \longleftrightarrow w \in \text{Deriv } a \ L$

**by** (*metis Cons.IH bisimilar.cases*)

**thus** *?case* **by** (*auto simp: Deriv-def*)

**next**

**assume**  $a \notin S$

**thus** *?case* **using** *Cons.prem*s **by** *auto*

**qed**

**qed**

**qed**

**lemma** *language-coinduct*:

**fixes**  $R$  (*infixl*  $\sim$  50)

**assumes**  $\bigwedge K \ L. K \sim L \implies K \subseteq \text{lists } S \wedge L \subseteq \text{lists } S$

**assumes**  $K \sim L$

**assumes**  $\bigwedge K \ L. K \sim L \implies (\square \in K \longleftrightarrow \square \in L)$

**assumes**  $\bigwedge K \ L \ x. K \sim L \implies x : S \implies \text{Deriv } x \ K \sim \text{Deriv } x \ L$

**shows**  $K = L$

**apply** (*rule equal-if-bisimilar*)

**apply** (*metis assms(1) assms(2)*)

**apply** (*metis assms(1) assms(2)*)

**apply** (*rule bisimilar.coinduct[of R, OF  $\langle K \sim L \rangle$ ]*)

**apply** (*auto simp: assms*)

**done**

**end**

**type-synonym** *rexp-pair* = *nat rexp \* nat rexp*

**type-synonym** *rexp-pairs* = *rexp-pair list*

**definition** *is-bisimulation* :: *nat list*  $\Rightarrow$  *rexp-pairs*  $\Rightarrow$  *bool*

**where**

*is-bisimulation* *as ps* =

$(\forall (r,s) \in \text{set } ps. (\text{atoms } r \cup \text{atoms } s \subseteq \text{set } as) \wedge (\text{nullable } r \longleftrightarrow \text{nullable } s) \wedge$   
 $(\forall a \in \text{set } as. (\text{nderiv } a \ r, \text{nderiv } a \ s) \in \text{set } ps))$

**lemma** *bisim-lang-eq*:

**assumes** *bisim*: *is-bisimulation as ps*

**assumes**  $\langle r, s \rangle \in \text{set } ps$

**shows**  $\text{lang } (\text{set } as) \ r = \text{lang } (\text{set } as) \ s$

**proof** –

**let**  $?R = \lambda K \ L. (\exists (r,s) \in \text{set } ps. K = \text{lang } (\text{set } as) \ r \wedge L = \text{lang } (\text{set } as) \ s)$

**show** *?thesis*

**proof** (*rule language-coinduct*[**where**  $R=?R$  **and**  $S = \text{set } as$ ])

**from**  $\langle r, s \rangle \in \text{set } ps$  **show**  $?R (\text{lang } (\text{set } as) \ r) (\text{lang } (\text{set } as) \ s)$

**by** *auto*

**next**

**fix**  $K \ L$  **assume**  $?R \ K \ L$

**then obtain**  $r \ s$  **where**  $rs: (r, s) \in \text{set } ps$

**and**  $KL: K = \text{lang } (\text{set } as) \ r \ L = \text{lang } (\text{set } as) \ s$  **by** *auto*

**with** *bisim* **have**  $\text{nullable } r \longleftrightarrow \text{nullable } s$

**by** (*auto simp: is-bisimulation-def*)

**thus**  $\square \in K \longleftrightarrow \square \in L$  **by** (*auto simp: nullable-iff*[**where**  $S=\text{set } as$ ]  $KL$ )

next case, but shared context

**from** *bisim rs KL lang-subset-lists*[*of - set as*]

**show**  $K \subseteq \text{lists } (\text{set } as) \wedge L \subseteq \text{lists } (\text{set } as)$

**unfolding** *is-bisimulation-def* **by** *blast*

next case, but shared context

**fix**  $a$  **assume**  $a \in \text{set } as$

**with**  $rs$  *bisim*

**have**  $(\text{nderiv } a \ r, \text{nderiv } a \ s) \in \text{set } ps$

**by** (*auto simp: is-bisimulation-def*)

**thus**  $?R (\text{Deriv } a \ K) (\text{Deriv } a \ L)$  **using**  $\langle a \in \text{set } as \rangle$

**by** (*force simp: KL lang-nderiv*)

**qed**

**qed**

## 11.5 Closure computation

**fun** *test* :: *rexp-pairs* \* *rexp-pairs*  $\Rightarrow$  *bool*

**where** *test* ( $ws, ps$ ) = (*case ws of*  $\square \Rightarrow \text{False} \mid (p,q)\#- \Rightarrow \text{nullable } p = \text{nullable } q$ )

**fun** *step* :: *nat list*  $\Rightarrow$  *rexp-pairs* \* *rexp-pairs*  $\Rightarrow$  *rexp-pairs* \* *rexp-pairs*

**where** *step as* ( $ws, ps$ ) =

(*let*

$(r, s) = \text{hd } ws;$

$ps' = (r, s) \# ps;$

$\text{succs} = \text{map } (\lambda a. (\text{nderiv } a \ r, \text{nderiv } a \ s)) \ as;$

$\text{new} = \text{filter } (\lambda p. p \notin \text{set } ps' \cup \text{set } ws) \ \text{succs}$

*in (new @ tl ws, ps')*

**definition** *closure* ::

*nat list*  $\Rightarrow$  *rexp-pairs* \* *rexp-pairs*  
 $\Rightarrow$  (*rexp-pairs* \* *rexp-pairs*) *option* **where**  
*closure as* = *while-option test* (*step as*)

**definition** *pre-bisim* :: *nat list*  $\Rightarrow$  *nat rexp*  $\Rightarrow$  *nat rexp*  $\Rightarrow$   
*rexp-pairs* \* *rexp-pairs*  $\Rightarrow$  *bool*

**where**

*pre-bisim as r s* = ( $\lambda(ws,ps)$ ).  
 $((r, s) \in \text{set } ws \cup \text{set } ps) \wedge$   
 $(\forall (r,s) \in \text{set } ws \cup \text{set } ps. \text{atoms } r \cup \text{atoms } s \subseteq \text{set } as) \wedge$   
 $(\forall (r,s) \in \text{set } ps. (\text{nullable } r \longleftrightarrow \text{nullable } s) \wedge$   
 $(\forall a \in \text{set } as. (\text{nderiv } a \ r, \text{nderiv } a \ s) \in \text{set } ps \cup \text{set } ws)))$

**theorem** *closure-sound*:

**assumes** *result*: *closure as* ( $[(r,s)], []$ ) = *Some*( $[], ps$ )

**and** *atoms*: *atoms r*  $\cup$  *atoms s*  $\subseteq$  *set as*

**shows** *lang* (*set as*) *r* = *lang* (*set as*) *s*

**proof** –

{ **fix** *st* **have** *pre-bisim as r s st*  $\implies$  *test st*  $\implies$  *pre-bisim as r s* (*step as st*)  
**unfolding** *pre-bisim-def*  
**by** (*cases st*) (*auto simp: split-def split: list.splits prod.splits*  
*dest!: subsetD[OF atoms-nderiv]*) }

**moreover**

**from** *atoms*

**have** *pre-bisim as r s* ( $[(r,s)], []$ ) **by** (*simp add: pre-bisim-def*)

**ultimately have** *pre-bisim-ps*: *pre-bisim as r s* ( $[], ps$ )

**by** (*rule while-option-rule[OF - result[unfolded closure-def]]*)

**then have** *is-bisimulation as ps* (*r, s*)  $\in$  *set ps*

**by** (*auto simp: pre-bisim-def is-bisimulation-def*)

**thus** *lang* (*set as*) *r* = *lang* (*set as*) *s* **by** (*rule bisim-lang-eq*)

**qed**

## 11.6 The overall procedure

**primrec** *add-atoms* :: *nat rexp*  $\Rightarrow$  *nat list*  $\Rightarrow$  *nat list*

**where**

*add-atoms Zero* = *id*  
| *add-atoms One* = *id*  
| *add-atoms (Atom a)* = *List.insert a*  
| *add-atoms (Plus r s)* = *add-atoms s o add-atoms r*  
| *add-atoms (Times r s)* = *add-atoms s o add-atoms r*  
| *add-atoms (Not r)* = *add-atoms r*  
| *add-atoms (Inter r s)* = *add-atoms s o add-atoms r*  
| *add-atoms (Star r)* = *add-atoms r*

**lemma** *set-add-atoms*: *set* (*add-atoms r as*) = *atoms r*  $\cup$  *set as*

**by** (*induct r arbitrary: as*) *auto*

**definition** *check-equiv* :: *nat list*  $\Rightarrow$  *nat rexp*  $\Rightarrow$  *nat rexp*  $\Rightarrow$  *bool*

**where**

*check-equiv as r s*  $\longleftrightarrow$  *set*(*add-atoms r* (*add-atoms s* []))  $\subseteq$  *set as*  $\wedge$   
(*case closure as* ([[*norm r*, *norm s*]], [])) *of*  
    *Some*([],-)  $\Rightarrow$  *True* | -  $\Rightarrow$  *False*

**lemma** *soundness*:

**assumes** *check-equiv as r s* **shows** *lang* (*set as*) *r* = *lang* (*set as*) *s*

**proof** –

**obtain** *ps* **where** *cl*: *closure as* ([[*norm r*,*norm s*]], []) = *Some*([],*ps*)

**and** *at*: *atoms r*  $\cup$  *atoms s*  $\subseteq$  *set as*

**using** *assms*

**by** (*auto simp: check-equiv-def set-add-atoms split:option.splits list.splits*)

**hence** *atoms*(*norm r*)  $\cup$  *atoms*(*norm s*)  $\subseteq$  *set as*

**using** *atoms-norm* **by** *blast*

**hence** *lang* (*set as*) (*norm r*) = *lang* (*set as*) (*norm s*)

**by** (*rule closure-sound[OF cl]*)

**thus** *lang* (*set as*) *r* = *lang* (*set as*) *s* **by** *simp*

**qed**

**lemma** *check-equiv* [0] (*Plus One* (*Times* (*Atom* 0) (*Star*(*Atom* 0)))) (*Star*(*Atom* 0))

**by** *eval*

**lemma** *check-equiv* [0,1] (*Not*(*Atom* 0))

    (*Plus One* (*Times* (*Plus* (*Atom* 1) (*Times* (*Atom* 0) (*Plus* (*Atom* 0) (*Atom* 1))))  
    (*Star*(*Plus* (*Atom* 0) (*Atom* 1))))))

**by** *eval*

**lemma** *check-equiv* [0] (*Atom* 0) (*Inter* (*Star* (*Atom* 0)) (*Atom* 0))

**by** *eval*

**end**

## References

- [1] V. Antimirov. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science*, 155:291–319, 1995.
- [2] J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11:481–494, 1964.
- [3] A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Automated Reasoning*, 49:95–106, 2012. published online March 2011.