

# tphols-2011

By xingyuan

January 26, 2011

## Contents

<b>1</b>	<b>Preliminary definitions</b>	<b>1</b>
<b>2</b>	<b>Direction <i>finite partition</i> <math>\Rightarrow</math> <i>regular language</i></b>	<b>5</b>
2.1	The proof of this direction . . . . .	10
2.1.1	Basic properties . . . . .	10
2.1.2	Intialization . . . . .	11
2.1.3	Interation step . . . . .	13
2.1.4	Conclusion of the proof . . . . .	19
<b>3</b>	<b>Direction: <i>regular language</i> <math>\Rightarrow</math> <i>finite partition</i></b>	<b>21</b>
3.1	The scheme for this direction . . . . .	21
3.2	Lemmas for basic cases . . . . .	22
3.3	The case for <i>SEQ</i> . . . . .	23
3.4	The case for <i>ALT</i> . . . . .	25
3.5	The case for <i>STAR</i> . . . . .	26
3.6	The main lemma . . . . .	28

```
theory Prelude
imports Main
begin
```

```
lemma set-eq-intro:
   $(\bigwedge x. (x \in A) = (x \in B)) \implies A = B$ 
by blast
```

```
end
theory Myhill
  imports Main List-Prefix Prefix-subtract Prelude
begin
```

# 1 Preliminary definitions

Sequential composition of two languages  $L1$  and  $L2$

**definition**  $Seq :: string\ set \Rightarrow string\ set \Rightarrow string\ set$  (- ;; - [100,100] 100)  
**where**

$$L1 ;; L2 = \{s1 @ s2 \mid s1\ s2. s1 \in L1 \wedge s2 \in L2\}$$

Transitive closure of language  $L$ .

**inductive-set**

$Star :: string\ set \Rightarrow string\ set$  (-★ [101] 102)

**for**  $L :: string\ set$

**where**

$start[intro]: [] \in L\star$

$| step[intro]: [s1 \in L; s2 \in L\star] \Longrightarrow s1@s2 \in L\star$

Some properties of operator ;;.

**lemma** *seq-union-distrib*:

$$(A \cup B) ;; C = (A ;; C) \cup (B ;; C)$$

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

**lemma** *seq-intro*:

$$[x \in A; y \in B] \Longrightarrow x @ y \in A ;; B$$

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

**lemma** *seq-assoc*:

$$(A ;; B) ;; C = A ;; (B ;; C)$$

**apply** (*auto simp:Seq-def*)

**apply** *blast*

**by** (*metis append-assoc*)

**lemma** *star-intro1[rule-format]*:  $x \in lang\star \Longrightarrow \forall y. y \in lang\star \longrightarrow x @ y \in lang\star$

**by** (*erule Star.induct, auto*)

**lemma** *star-intro2*:  $y \in lang \Longrightarrow y \in lang\star$

**by** (*drule step[of y lang []], auto simp:start*)

**lemma** *star-intro3[rule-format]*:

$$x \in lang\star \Longrightarrow \forall y. y \in lang \longrightarrow x @ y \in lang\star$$

**by** (*erule Star.induct, auto intro:star-intro2*)

**lemma** *star-decom*:

$$[x \in lang\star; x \neq []] \Longrightarrow (\exists a\ b. x = a @ b \wedge a \neq [] \wedge a \in lang \wedge b \in lang\star)$$

**by** (*induct x rule: Star.induct, simp, blast*)

**lemma** *star-decom'*:

$$[x \in lang\star; x \neq []] \Longrightarrow \exists a\ b. x = a @ b \wedge a \in lang\star \wedge b \in lang$$

**apply** (*induct x rule:Star.induct, simp*)

**apply** (*case-tac s2 = []*)

```

apply (rule-tac x = [] in exI, rule-tac x = s1 in exI, simp add:start)
apply (simp, (erule exE | erule conjE)+)
by (rule-tac x = s1 @ a in exI, rule-tac x = b in exI, simp add:step)

```

Ardens lemma expressed at the level of language, rather than the level of regular expression.

**theorem** *ardens-revised*:

```

assumes nemp: []  $\notin$  A
shows (X = X ;; A  $\cup$  B)  $\longleftrightarrow$  (X = B ;; A $\star$ )

```

**proof**

```

assume eq: X = B ;; A $\star$ 
have A $\star$  = {[]}  $\cup$  A $\star$  ;; A
  by (auto simp:Seq-def star-intro3 star-decom $\wedge$ )
then have B ;; A $\star$  = B ;; ({}  $\cup$  A $\star$  ;; A)
  unfolding Seq-def by simp
also have ... = B  $\cup$  B ;; (A $\star$  ;; A)
  unfolding Seq-def by auto
also have ... = B  $\cup$  (B ;; A $\star$ ) ;; A
  by (simp only:seq-assoc)
finally show X = X ;; A  $\cup$  B
  using eq by blast

```

**next**

```

assume eq': X = X ;; A  $\cup$  B
hence c1':  $\bigwedge$  x. x  $\in$  B  $\implies$  x  $\in$  X
  and c2':  $\bigwedge$  x y. [x  $\in$  X; y  $\in$  A]  $\implies$  x @ y  $\in$  X
  using Seq-def by auto
show X = B ;; A $\star$ 

```

**proof**

```

show B ;; A $\star$   $\subseteq$  X
proof-
  { fix x y
    have [y  $\in$  A $\star$ ; x  $\in$  X]  $\implies$  x @ y  $\in$  X
      apply (induct arbitrary:x rule:Star.induct, simp)
      by (auto simp only:append-assoc[THEN sym] dest:c2 $\wedge$ )
    } thus ?thesis using c1' by (auto simp:Seq-def)

```

**qed**

**next**

```

show X  $\subseteq$  B ;; A $\star$ 
proof-
  { fix x
    have x  $\in$  X  $\implies$  x  $\in$  B ;; A $\star$ 
      proof (induct x taking:length rule:measure-induct)
        fix z
        assume hyps:
           $\forall$  y. length y < length z  $\longrightarrow$  y  $\in$  X  $\longrightarrow$  y  $\in$  B ;; A $\star$ 
          and z-in: z  $\in$  X
        show z  $\in$  B ;; A $\star$ 
        proof (cases z  $\in$  B)
          case True thus ?thesis by (auto simp:Seq-def start)

```

```

next
  case False hence  $z \in X$  ;; A using eq' z-in by auto
  then obtain za zb where za-in:  $za \in X$ 
    and zab:  $z = za @ zb \wedge zb \in A$  and zbne:  $zb \neq []$ 
    using nemp unfolding Seq-def by blast
  from zbne zab have  $\text{length } za < \text{length } z$  by auto
  with za-in hyps have  $za \in B$  ;; A* by blast
  hence  $za @ zb \in B$  ;; A* using zab
    by (clarsimp simp:Seq-def, blast dest:star-intro3)
  thus ?thesis using zab by simp
qed
qed
} thus ?thesis by blast
qed
qed
qed

```

The syntax of regular expressions is defined by the datatype *rexp*.

```

datatype rexp =
  NULL
| EMPTY
| CHAR char
| SEQ rexp rexp
| ALT rexp rexp
| STAR rexp

```

The following *L* is an overloaded operator, where  $L(x)$  evaluates to the language represented by the syntactic object *x*.

```

consts L:: 'a  $\Rightarrow$  string set

```

The  $L(\textit{rexp})$  for regular expression *rexp* is defined by the following overloading function *L-rexp*.

```

overloading L-rexp  $\equiv$  L:: rexp  $\Rightarrow$  string set
begin
fun
  L-rexp :: rexp  $\Rightarrow$  string set
where
  L-rexp (NULL) = {}
| L-rexp (EMPTY) = {}
| L-rexp (CHAR c) = {[c]}
| L-rexp (SEQ r1 r2) = (L-rexp r1) ;; (L-rexp r2)
| L-rexp (ALT r1 r2) = (L-rexp r1)  $\cup$  (L-rexp r2)
| L-rexp (STAR r) = (L-rexp r)*
end

```

To obtain equational system out of finite set of equivalent classes, a fold operation on finite set *folds* is defined. The use of *SOME* makes *fold* more robust than the *fold* in Isabelle library. The expression *folds f* makes sense when *f* is not *associative* and *commutitive*, while *fold f* does not.

**definition**

$$folds :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ set} \Rightarrow 'b$$
**where**

$$folds f z S \equiv \text{SOME } x. \text{ fold-graph } f z S x$$

The following lemma assures that the arbitrary choice made by the *SOME* in *folds* does not affect the *L*-value of the resultant regular expression.

**lemma** *folds-alt-simp* [*simp*]:
$$\text{finite } rs \implies L (\text{folds } ALT \text{ NULL } rs) = \bigcup (L \text{ ` } rs)$$
**apply** (*rule set-eq-intro*, *simp add:folds-def*)**apply** (*rule someI2-ex*, *erule finite-imp-fold-graph*)**by** (*erule fold-graph.induct*, *auto*)**lemma** [*simp*]:
$$\text{shows } (x, y) \in \{(x, y). P x y\} \longleftrightarrow P x y$$
**by** *simp*

$\approx L$  is an equivalent class defined by language *Lang*.

**definition**

$$\text{str-eq-rel } (\approx -)$$
**where**

$$\approx Lang \equiv \{(x, y). (\forall z. x @ z \in Lang \longleftrightarrow y @ z \in Lang)\}$$

Among equivalent classes of  $\approx Lang$ , the set *finals*(*Lang*) singles out those which contains strings from *Lang*.

**definition**

$$\text{finals } Lang \equiv \{\approx Lang \text{ ` } \{x\} \mid x . x \in Lang\}$$

The following lemma show the relationship between *finals*(*Lang*) and *Lang*.

**lemma** *lang-is-union-of-finals*:
$$Lang = \bigcup \text{finals}(Lang)$$
**proof****show**  $Lang \subseteq \bigcup (\text{finals } Lang)$ **proof****fix** *x***assume**  $x \in Lang$ **thus**  $x \in \bigcup (\text{finals } Lang)$ **apply** (*simp add:finals-def*, *rule-tac x = (\approx Lang) ` {x} in exI*)**by** (*auto simp:Image-def str-eq-rel-def*)**qed****next****show**  $\bigcup (\text{finals } Lang) \subseteq Lang$ **apply** (*clarsimp simp:finals-def str-eq-rel-def*)**by** (*drule-tac x = [] in spec*, *auto*)**qed**

## 2 Direction *finite partition* $\Rightarrow$ *regular language*

The relationship between equivalent classes can be described by an equational system. For example, in equational system (1),  $X_0, X_1$  are equivalent classes. The first equation says every string in  $X_0$  is obtained either by appending one  $b$  to a string in  $X_0$  or by appending one  $a$  to a string in  $X_1$  or just be an empty string (represented by the regular expression  $\lambda$ ). Similarly, the second equation tells how the strings inside  $X_1$  are composed.

$$\begin{aligned} X_0 &= X_0b + X_1a + \lambda \\ X_1 &= X_0a + X_1b \end{aligned} \tag{1}$$

The summands on the right hand side is represented by the following datatype *rhs-item*, mnemonic for 'right hand side item'. Generally, there are two kinds of right hand side items, one kind corresponds to pure regular expressions, like the  $\lambda$  in (1), the other kind corresponds to transitions from one one equivalent class to another, like the  $X_0b, X_1a$  etc.

```
datatype rhs-item =
  Lam rexp
| Trn (string set) rexp
```

In this formalization, pure regular expressions like  $\lambda$  is represented by *Lam*(*EMPTY*), while transitions like  $X_0a$  is represented by *Trn*  $X_0$  (*CHAR*  $a$ ).

The functions *the-r* and *the-Trn* are used to extract subcomponents from right hand side items.

```
fun the-r :: rhs-item  $\Rightarrow$  rexp
where the-r (Lam  $r$ ) =  $r$ 
```

```
fun the-Trn:: rhs-item  $\Rightarrow$  (string set  $\times$  rexp)
where the-Trn (Trn  $Y$   $r$ ) = ( $Y$ ,  $r$ )
```

Every right hand side item *itm* defines a string set given  $L(itm)$ , defined as:

```
overloading L-rhs-e  $\equiv$  L:: rhs-item  $\Rightarrow$  string set
begin
  fun L-rhs-e:: rhs-item  $\Rightarrow$  string set
  where
    L-rhs-e (Lam  $r$ ) = L  $r$  |
    L-rhs-e (Trn  $X$   $r$ ) =  $X$  ;; L  $r$ 
end
```

The right hand side of every equation is represented by a set of items. The string set defined by such a set *itms* is given by  $L(itms)$ , defined as:

```
overloading L-rhs  $\equiv$  L:: rhs-item set  $\Rightarrow$  string set
begin
  fun L-rhs:: rhs-item set  $\Rightarrow$  string set
```

**where**  $L\text{-rhs } rhs = \bigcup (L \text{ ' } rhs)$   
**end**

Given a set of equivalent classes  $CS$  and one equivalent class  $X$  among  $CS$ , the term  $init\text{-rhs } CS X$  is used to extract the right hand side of the equation describing the formation of  $X$ . The definition of  $init\text{-rhs}$  is:

**definition**

$init\text{-rhs } CS X \equiv$   
 if  $([] \in X)$  then  
 $\{Lam(EMPTY)\} \cup \{Trn Y (CHAR c) \mid Y c. Y \in CS \wedge Y ;; \{[c]\} \subseteq X\}$   
 else  
 $\{Trn Y (CHAR c) \mid Y c. Y \in CS \wedge Y ;; \{[c]\} \subseteq X\}$

In the definition of  $init\text{-rhs}$ , the term  $\{Trn Y (CHAR c) \mid Y c. Y \in CS \wedge Y ;; \{[c]\} \subseteq X\}$  appearing on both branches describes the formation of strings in  $X$  out of transitions, while the term  $\{Lam(EMPTY)\}$  describes the empty string which is intrinsically contained in  $X$  rather than by transition. This  $\{Lam(EMPTY)\}$  corresponds to the  $\lambda$  in (1).

With the help of  $init\text{-rhs}$ , the equational system describing the formation of every equivalent class inside  $CS$  is given by the following  $eqs(CS)$ .

**definition**  $eqs CS \equiv \{(X, init\text{-rhs } CS X) \mid X. X \in CS\}$

The following  $items\text{-of } rhs X$  returns all  $X$ -items in  $rhs$ .

**definition**

$items\text{-of } rhs X \equiv \{Trn X r \mid r. (Trn X r) \in rhs\}$

The following  $rexp\text{-of } rhs X$  combines all regular expressions in  $X$ -items using  $ALT$  to form a single regular expression. It will be used later to implement  $arden\text{-variate}$  and  $rhs\text{-subst}$ .

**definition**

$rexp\text{-of } rhs X \equiv folds ALT NULL ((snd o the\text{-Trn}) \text{ ' } items\text{-of } rhs X)$

The following  $lam\text{-of } rhs$  returns all pure regular expression items in  $rhs$ .

**definition**

$lam\text{-of } rhs \equiv \{Lam r \mid r. Lam r \in rhs\}$

The following  $rexp\text{-of-lam } rhs$  combines pure regular expression items in  $rhs$  using  $ALT$  to form a single regular expression. When all variables inside  $rhs$  are eliminated,  $rexp\text{-of-lam } rhs$  is used to compute the regular expression corresponds to  $rhs$ .

**definition**

$rexp\text{-of-lam } rhs \equiv folds ALT NULL (the\text{-r} \text{ ' } lam\text{-of } rhs)$

The following  $attach\text{-rexp } rexp' itm$  attach the regular expression  $rexp'$  to the right of right hand side item  $itm$ .

**fun**  $attach\text{-rexp} :: rexp \Rightarrow rhs\text{-item} \Rightarrow rhs\text{-item}$

**where**

$$\begin{aligned} \text{attach-}r\text{exp } r\text{exp}' (Lam \ r\text{exp}) &= Lam \ (SEQ \ r\text{exp } r\text{exp}') \\ | \text{attach-}r\text{exp } r\text{exp}' (Trn \ X \ r\text{exp}) &= Trn \ X \ (SEQ \ r\text{exp } r\text{exp}') \end{aligned}$$

The following *append-rhs-rxp*  $r\text{hs } r\text{exp}$  attaches  $r\text{exp}$  to every item in  $r\text{hs}$ .

**definition**

$$\text{append-rhs-rxp } r\text{hs } r\text{exp} \equiv (\text{attach-rxp } r\text{exp}) \text{ ' } r\text{hs}$$

With the help of the two functions immediately above, Ardens' transformation on right hand side  $r\text{hs}$  is implemented by the following function *arden-variate*  $X \ r\text{hs}$ . After this transformation, the recursive occurent of  $X$  in  $r\text{hs}$  will be eliminated, while the string set defined by  $r\text{hs}$  is kept unchanged.

**definition**

$$\begin{aligned} \text{arden-variate } X \ r\text{hs} &\equiv \\ \text{append-rhs-rxp } (r\text{hs} - \text{items-of } r\text{hs } X) &(\text{STAR } (\text{rxp-of } r\text{hs } X)) \end{aligned}$$

Suppose the equation defining  $X$  is  $X = xr\text{hs}$ , the purpose of *rhs-subst* is to substitute all occurrences of  $X$  in  $r\text{hs}$  by  $xr\text{hs}$ . A little thought may reveal that the final result should be: first append  $(a_1|a_2|\dots|a_n)$  to every item of  $xr\text{hs}$  and then union the result with all non- $X$ -items of  $r\text{hs}$ .

**definition**

$$\begin{aligned} \text{rhs-subst } r\text{hs } X \ xr\text{hs} &\equiv \\ (r\text{hs} - (\text{items-of } r\text{hs } X)) \cup &(\text{append-rhs-rxp } xr\text{hs } (\text{rxp-of } r\text{hs } X)) \end{aligned}$$

Suppose the equation defining  $X$  is  $X = xr\text{hs}$ , the following *eqs-subst*  $ES \ X \ xr\text{hs}$  substitute  $xr\text{hs}$  into every equation of the equational system  $ES$ .

**definition**

$$\text{eqs-subst } ES \ X \ xr\text{hs} \equiv \{(Y, \text{rhs-subst } yr\text{hs } X \ xr\text{hs}) \mid Y \ yr\text{hs}. (Y, \ yr\text{hs}) \in ES\}$$

The computation of regular expressions for equivalent classes is accomplished using a iteration principle given by the following lemma.

**lemma** *wf-iter* [*rule-format*]:

**fixes**  $f$

**assumes** *step*:  $\bigwedge e. \llbracket P \ e; \neg Q \ e \rrbracket \implies (\exists e'. P \ e' \wedge (f(e'), f(e)) \in \text{less-than})$

**shows** *pe*:  $P \ e \longrightarrow (\exists e'. P \ e' \wedge Q \ e')$

**proof**(*induct e rule: wf-induct*

[*OF wf-inv-image*[*OF wf-less-than*, **where**  $f = f$ ]], *clarify*)

**fix**  $x$

**assume**  $h$  [*rule-format*]:

$\forall y. (y, x) \in \text{inv-image less-than } f \longrightarrow P \ y \longrightarrow (\exists e'. P \ e' \wedge Q \ e')$

**and**  $px: P \ x$

**show**  $\exists e'. P \ e' \wedge Q \ e'$

**proof**(*cases Q x*)

**assume**  $Q \ x$  **with**  $px$  **show** *?thesis* **by** *blast*

**next**

**assume**  $nq: \neg Q \ x$



```

from step [OF px nq]
obtain e' where pe': P e' and ltf: (f e', f x) ∈ less-than by auto
show ?thesis
proof(rule h)
  from ltf show (e', x) ∈ inv-image less-than f
  by (simp add:inv-image-def)
next
  from pe' show P e' .
qed
qed
qed

```

The  $P$  in lemma *wf-iter* is an invariant kept throughout the iteration procedure. The particular invariant used to solve our problem is defined by function  $Inv(ES)$ , an invariant over equal system  $ES$ . Every definition starting next till  $Inv$  stipulates a property to be satisfied by  $ES$ .

Every variable is defined at most once in  $ES$ .

**definition**

$$\text{distinct-equas } ES \equiv \forall X \text{ rhs rhs}'. (X, \text{rhs}) \in ES \wedge (X, \text{rhs}') \in ES \longrightarrow \text{rhs} = \text{rhs}'$$

Every equation in  $ES$  (represented by  $(X, \text{rhs})$ ) is valid, i.e.  $(X = L \text{ rhs})$ .

**definition**

$$\text{valid-eqns } ES \equiv \forall X \text{ rhs}. (X, \text{rhs}) \in ES \longrightarrow (X = L \text{ rhs})$$

The following *rhs-nonempty rhs* requires regular expressions occurring in transitional items of  $\text{rhs}$  does not contain empty string. This is necessary for the application of Arden's transformation to  $\text{rhs}$ .

**definition**

$$\text{rhs-nonempty rhs} \equiv (\forall Y r. \text{Trn } Y r \in \text{rhs} \longrightarrow [] \notin L r)$$

The following *ardenable ES* requires that Arden's transformation is applicable to every equation of equational system  $ES$ .

**definition**

$$\text{ardenable } ES \equiv \forall X \text{ rhs}. (X, \text{rhs}) \in ES \longrightarrow \text{rhs-nonempty rhs}$$

**definition**

$$\text{non-empty } ES \equiv \forall X \text{ rhs}. (X, \text{rhs}) \in ES \longrightarrow X \neq \{\}$$

The following *finite-rhs ES* requires every equation in  $\text{rhs}$  be finite.

**definition**

$$\text{finite-rhs } ES \equiv \forall X \text{ rhs}. (X, \text{rhs}) \in ES \longrightarrow \text{finite rhs}$$

The following *classes-of rhs* returns all variables (or equivalent classes) occurring in  $\text{rhs}$ .

**definition**

$$\text{classes-of } rhs \equiv \{X. \exists r. \text{Trn } X r \in rhs\}$$

The following *lefts-of ES* returns all variables defined by equational system *ES*.

**definition**

$$\text{lefts-of } ES \equiv \{Y \mid Y \text{ yrhs. } (Y, \text{yrhs}) \in ES\}$$

The following *self-contained ES* requires that every variable occurring on the right hand side of equations is already defined by some equation in *ES*.

**definition**

$$\text{self-contained } ES \equiv \forall (X, \text{rhrs}) \in ES. \text{classes-of } \text{rhrs} \subseteq \text{lefts-of } ES$$

The invariant  $\text{Inv}(ES)$  is a conjunction of all the previously defined constraints.

**definition**

$$\text{Inv } ES \equiv \text{valid-egns } ES \wedge \text{finite } ES \wedge \text{distinct-equas } ES \wedge \text{ardenable } ES \wedge \\ \text{non-empty } ES \wedge \text{finite-rhs } ES \wedge \text{self-contained } ES$$

## 2.1 The proof of this direction

### 2.1.1 Basic properties

The following are some basic properties of the above definitions.

**lemma** *L-rhs-union-distrib*:

$$L (A::\text{rhs-item set}) \cup L B = L (A \cup B)$$

by *simp*

**lemma** *finite-snd-Trn*:

assumes *finite:finite rhs*

shows *finite*  $\{r_2. \text{Trn } Y r_2 \in rhs\}$  (is *finite ?B*)

**proof** –

**def**  $rhs' \equiv \{e \in rhs. \exists r. e = \text{Trn } Y r\}$

**have**  $?B = (\text{snd o the-Trn}) \text{ ` } rhs'$  **using** *rhs'-def* **by** (*auto simp:image-def*)

**moreover have** *finite rhs'* **using** *finite rhs'-def* **by** *auto*

**ultimately show** *?thesis* **by** *simp*

**qed**

**lemma** *rexp-of-empty*:

assumes *finite:finite rhs*

**and** *nonempty:rhs-nonempty rhs*

shows  $\square \notin L (\text{rexp-of } rhs X)$

**using** *finite nonempty rhs-nonempty-def*

**by** (*drule-tac finite-snd-Trn[where Y = X], auto simp:rexp-of-def items-of-def*)

**lemma** [*intro!*]:

$$P (\text{Trn } X r) \implies (\exists a. (\exists r. a = \text{Trn } X r \wedge P a)) \text{ by } \textit{auto}$$

**lemma** *finite-items-of*:  
 $finite\ rhs \implies finite\ (items\ of\ rhs\ X)$   
**by** (*auto simp:items-of-def intro:finite-subset*)

**lemma** *lang-of-rexp-of*:  
**assumes** *finite:finite rhs*  
**shows**  $L\ (items\ of\ rhs\ X) = X \ ;\ ;\ (L\ (rexp\ of\ rhs\ X))$   
**proof** –  
**have**  $finite\ ((snd\ \circ\ the\ Trn)\ 'items\ of\ rhs\ X)$  **using** *finite-items-of[OF finite]*  
**by** *auto*  
**thus** *?thesis*  
**apply** (*auto simp:rexp-of-def Seq-def items-of-def*)  
**apply** (*rule-tac x = s1 in exI, rule-tac x = s2 in exI, auto*)  
**by** (*rule-tac x = Trn X r in exI, auto simp:Seq-def*)  
**qed**

**lemma** *rexp-of-lam-eq-lam-set*:  
**assumes** *finite: finite rhs*  
**shows**  $L\ (rexp\ of\ lam\ rhs) = L\ (lam\ of\ rhs)$   
**proof** –  
**have**  $finite\ (the\ r\ ' \{Lam\ r\ |r.\ Lam\ r\ \in\ rhs\})$  **using** *finite*  
**by** (*rule-tac finite-imageI, auto intro:finite-subset*)  
**thus** *?thesis* **by** (*auto simp:rexp-of-lam-def lam-of-def*)  
**qed**

**lemma** [*simp*]:  
 $L\ (attach\ rexp\ r\ xb) = L\ xb \ ;\ ;\ L\ r$   
**apply** (*cases xb, auto simp:Seq-def*)  
**by** (*rule-tac x = s1 @ s1a in exI, rule-tac x = s2a in exI, auto simp:Seq-def*)

**lemma** *lang-of-append-rhs*:  
 $L\ (append\ rhs\ rexp\ rhs\ r) = L\ rhs \ ;\ ;\ L\ r$   
**apply** (*auto simp:append-rhs-rexp-def image-def*)  
**apply** (*auto simp:Seq-def*)  
**apply** (*rule-tac x = L xb ;; L r in exI, auto simp add:Seq-def*)  
**by** (*rule-tac x = attach-rexp r xb in exI, auto simp:Seq-def*)

**lemma** *classes-of-union-distrib*:  
 $classes\ of\ A \cup\ classes\ of\ B = classes\ of\ (A \cup B)$   
**by** (*auto simp add:classes-of-def*)

**lemma** *lefts-of-union-distrib*:  
 $lefts\ of\ A \cup\ lefts\ of\ B = lefts\ of\ (A \cup B)$   
**by** (*auto simp:lefts-of-def*)

### 2.1.2 Intialization

The following several lemmas until *init-ES-satisfy-Inv* shows that the initial equational system satisfies invariant *Inv*.

**lemma** *defined-by-str*:

$\llbracket s \in X; X \in UNIV // (\approx Lang) \rrbracket \implies X = (\approx Lang) \text{ `` } \{s\}$   
**by** (*auto simp:quotient-def Image-def str-eq-rel-def*)

**lemma** *every-eclass-has-transition*:

**assumes** *has-str*:  $s @ [c] \in X$   
**and** *in-CS*:  $X \in UNIV // (\approx Lang)$   
**obtains**  $Y$  **where**  $Y \in UNIV // (\approx Lang)$  **and**  $Y ;; \{[c]\} \subseteq X$  **and**  $s \in Y$   
**proof** –  
**def**  $Y \equiv (\approx Lang) \text{ `` } \{s\}$   
**have**  $Y \in UNIV // (\approx Lang)$   
**unfolding** *Y-def quotient-def* **by** *auto*  
**moreover**  
**have**  $X = (\approx Lang) \text{ `` } \{s @ [c]\}$   
**using** *has-str in-CS defined-by-str* **by** *blast*  
**then have**  $Y ;; \{[c]\} \subseteq X$   
**unfolding** *Y-def Image-def Seq-def*  
**unfolding** *str-eq-rel-def*  
**by** *clarsimp*  
**moreover**  
**have**  $s \in Y$  **unfolding** *Y-def*  
**unfolding** *Image-def str-eq-rel-def* **by** *simp*  
**ultimately show thesis** **by** (*blast intro: that*)  
**qed**

**lemma** *l-eq-r-in-eqs*:

**assumes** *X-in-eqs*:  $(X, xrhs) \in (eqs (UNIV // (\approx Lang)))$   
**shows**  $X = L xrhs$   
**proof**  
**show**  $X \subseteq L xrhs$   
**proof**  
**fix**  $x$   
**assume** (1):  $x \in X$   
**show**  $x \in L xrhs$   
**proof** (*cases*  $x = []$ )  
**assume empty**:  $x = []$   
**thus ?thesis** **using** *X-in-eqs (1)*  
**by** (*auto simp:eqs-def init-rhs-def*)  
**next**  
**assume not-empty**:  $x \neq []$   
**then obtain** *clist c* **where** *decom*:  $x = clist @ [c]$   
**by** (*case-tac x rule:rev-cases, auto*)  
**have**  $X \in UNIV // (\approx Lang)$  **using** *X-in-eqs* **by** (*auto simp:eqs-def*)  
**then obtain**  $Y$   
**where**  $Y \in UNIV // (\approx Lang)$   
**and**  $Y ;; \{[c]\} \subseteq X$   
**and**  $clist \in Y$   
**using** *decom (1) every-eclass-has-transition* **by** *blast*  
**hence**

```

    x ∈ L {Trn Y (CHAR c) | Y c. Y ∈ UNIV // (≈Lang) ∧ Y ;; {[c]} ⊆ X}
  using (1) decom
  by (simp, rule-tac x = Trn Y (CHAR c) in exI, simp add:Seq-def)
  thus ?thesis using X-in-eqs (1)
  by (simp add:eqs-def init-rhs-def)
qed
qed
next
  show L xrhs ⊆ X using X-in-eqs
  by (auto simp:eqs-def init-rhs-def)
qed

```

**lemma** *finite-init-rhs*:

```

  assumes finite: finite CS
  shows finite (init-rhs CS X)
proof -
  have finite {Trn Y (CHAR c) | Y c. Y ∈ CS ∧ Y ;; {[c]} ⊆ X} (is finite ?A)
  proof -
    def S ≡ {(Y, c) | Y c. Y ∈ CS ∧ Y ;; {[c]} ⊆ X}
    def h ≡ λ (Y, c). Trn Y (CHAR c)
    have finite (CS × (UNIV::char set)) using finite by auto
    hence finite S using S-def
    by (rule-tac B = CS × UNIV in finite-subset, auto)
    moreover have ?A = h ` S by (auto simp: S-def h-def image-def)
    ultimately show ?thesis
    by auto
  qed
  thus ?thesis by (simp add:init-rhs-def)
qed

```

**lemma** *init-ES-satisfy-Inv*:

```

  assumes finite-CS: finite (UNIV // (≈Lang))
  shows Inv (eqs (UNIV // (≈Lang)))
proof -
  have finite (eqs (UNIV // (≈Lang))) using finite-CS
  by (simp add:eqs-def)
  moreover have distinct-equas (eqs (UNIV // (≈Lang)))
  by (simp add:distinct-equas-def eqs-def)
  moreover have ardenable (eqs (UNIV // (≈Lang)))
  by (auto simp add:ardenable-def eqs-def init-rhs-def rhs-nonempty-def del:L-rhs.simps)
  moreover have valid-eqns (eqs (UNIV // (≈Lang)))
  using l-eq-r-in-eqs by (simp add:valid-eqns-def)
  moreover have non-empty (eqs (UNIV // (≈Lang)))
  by (auto simp:non-empty-def eqs-def quotient-def Image-def str-eq-rel-def)
  moreover have finite-rhs (eqs (UNIV // (≈Lang)))
  using finite-init-rhs[OF finite-CS]
  by (auto simp:finite-rhs-def eqs-def)
  moreover have self-contained (eqs (UNIV // (≈Lang)))
  by (auto simp:self-contained-def eqs-def init-rhs-def classes-of-def lefts-of-def)

```

ultimately show *?thesis* by (simp add:Inv-def)  
qed

### 2.1.3 Iteration step

From this point until *iteration-step*, it is proved that there exists iteration steps which keep  $Inv(ES)$  while decreasing the size of  $ES$ .

**lemma** *arden-variate-keeps-eq*:

assumes *l-eq-r*:  $X = L\ rhs$   
and *not-empty*:  $\square \notin L\ (rexp\ of\ rhs\ X)$   
and *finite*: *finite rhs*  
shows  $X = L\ (arden\ variate\ X\ rhs)$

**proof** –

def  $A \equiv L\ (rexp\ of\ rhs\ X)$   
def  $b \equiv rhs - items\ of\ rhs\ X$   
def  $B \equiv L\ b$   
have  $X = B ;; A\star$

**proof** –

have  $rhs = items\ of\ rhs\ X \cup b$  by (auto simp:b-def items-of-def)  
hence  $L\ rhs = L(items\ of\ rhs\ X \cup b)$  by simp  
hence  $L\ rhs = L(items\ of\ rhs\ X) \cup B$  by (simp only:L-rhs-union-distrib B-def)  
with *lang-of-rexp-of*  
have  $L\ rhs = X ;; A \cup B$  using *finite* by (simp only:B-def b-def A-def)  
thus *?thesis*  
using *l-eq-r not-empty*  
apply (drule-tac  $B = B$  and  $X = X$  in *ardens-revised*)  
by (auto simp:A-def simp del:L-rhs.simps)

qed

moreover have  $L\ (arden\ variate\ X\ rhs) = (B ;; A\star)$  (is  $?L = ?R$ )  
by (simp only:arden-variate-def L-rhs-union-distrib lang-of-append-rhs  
B-def A-def b-def L-rexp.simps seq-union-distrib)

ultimately show *?thesis* by simp

qed

**lemma** *append-keeps-finite*:

*finite rhs*  $\implies$  *finite (append-rhs-rexp rhs r)*  
by (auto simp:append-rhs-rexp-def)

**lemma** *arden-variate-keeps-finite*:

*finite rhs*  $\implies$  *finite (arden-variate X rhs)*  
by (auto simp:arden-variate-def append-keeps-finite)

**lemma** *append-keeps-nonempty*:

*rhs-nonempty rhs*  $\implies$  *rhs-nonempty (append-rhs-rexp rhs r)*  
apply (auto simp:rhs-nonempty-def append-rhs-rexp-def)  
by (case-tac  $x$ , auto simp:Seq-def)

**lemma** *nonempty-set-sub*:

*rhs-nonempty rhs*  $\implies$  *rhs-nonempty (rhs - A)*

**by** (*auto simp:rhs-nonempty-def*)

**lemma** *nonempty-set-union*:

$\llbracket \text{rhs-nonempty } \text{rhs}; \text{rhs-nonempty } \text{rhs}' \rrbracket \implies \text{rhs-nonempty } (\text{rhs} \cup \text{rhs}')$

**by** (*auto simp:rhs-nonempty-def*)

**lemma** *arden-variate-keeps-nonempty*:

$\text{rhs-nonempty } \text{rhs} \implies \text{rhs-nonempty } (\text{arden-variate } X \text{ rhs})$

**by** (*simp only:arden-variate-def append-keeps-nonempty nonempty-set-sub*)

**lemma** *rhs-subst-keeps-nonempty*:

$\llbracket \text{rhs-nonempty } \text{rhs}; \text{rhs-nonempty } \text{xrhs} \rrbracket \implies \text{rhs-nonempty } (\text{rhs-subst } \text{rhs } X \text{ xrhs})$

**by** (*simp only:rhs-subst-def append-keeps-nonempty nonempty-set-union nonempty-set-sub*)

**lemma** *rhs-subst-keeps-eq*:

**assumes** *substor*:  $X = L \text{ xrhs}$

**and** *finite*: *finite rhs*

**shows**  $L (\text{rhs-subst } \text{rhs } X \text{ xrhs}) = L \text{ rhs}$  (**is** *?Left = ?Right*)

**proof** –

**def** *A*  $\equiv L (\text{rhs} - \text{items-of } \text{rhs } X)$

**have** *?Left*  $= A \cup L (\text{append-rhs-rexp } \text{xrhs} (\text{rexp-of } \text{rhs } X))$

**by** (*simp only:rhs-subst-def L-rhs-union-distrib A-def*)

**moreover have** *?Right*  $= A \cup L (\text{items-of } \text{rhs } X)$

**proof** –

**have**  $\text{rhs} = (\text{rhs} - \text{items-of } \text{rhs } X) \cup (\text{items-of } \text{rhs } X)$  **by** (*auto simp:items-of-def*)

**thus** *?thesis* **by** (*simp only:L-rhs-union-distrib A-def*)

**qed**

**moreover have**  $L (\text{append-rhs-rexp } \text{xrhs} (\text{rexp-of } \text{rhs } X)) = L (\text{items-of } \text{rhs } X)$

**using** *finite substor* **by** (*simp only:lang-of-append-rhs lang-of-rexp-of*)

**ultimately show** *?thesis* **by** *simp*

**qed**

**lemma** *rhs-subst-keeps-finite-rhs*:

$\llbracket \text{finite } \text{rhs}; \text{finite } \text{yrhs} \rrbracket \implies \text{finite } (\text{rhs-subst } \text{rhs } Y \text{ yrhs})$

**by** (*auto simp:rhs-subst-def append-keeps-finite*)

**lemma** *eqs-subst-keeps-finite*:

**assumes** *finite:finite* (*ES*::  $(\text{string set} \times \text{rhs-item set}) \text{ set}$ )

**shows** *finite* (*eqs-subst* *ES* *Y* *yrhs*)

**proof** –

**have** *finite*  $\{ (Ya, \text{rhs-subst } \text{yrhsa } Y \text{ yrhs}) \mid Ya \text{ yrhsa}. (Ya, \text{yrhsa}) \in \text{ES} \}$   
(**is** *finite ?A*)

**proof** –

**def** *eqns'*  $\equiv \{ ((Ya::\text{string set}), \text{yrhsa}) \mid Ya \text{ yrhsa}. (Ya, \text{yrhsa}) \in \text{ES} \}$

**def** *h*  $\equiv \lambda ((Ya::\text{string set}), \text{yrhsa}). (Ya, \text{rhs-subst } \text{yrhsa } Y \text{ yrhs})$

**have** *finite* (*h* ‘*eqns'*) **using** *finite h-def eqns'-def* **by** *auto*

**moreover have** *?A*  $= h$  ‘*eqns'* **by** (*auto simp:h-def eqns'-def*)

**ultimately show** *?thesis* **by** *auto*

**qed**  
**thus** *?thesis* **by** (*simp add: eqs-subst-def*)  
**qed**

**lemma** *eqs-subst-keeps-finite-rhs*:  
 $\llbracket \text{finite-rhs } ES; \text{finite } yrhs \rrbracket \implies \text{finite-rhs } (\text{eqs-subst } ES \ Y \ yrhs)$   
**by** (*auto intro: rhs-subst-keeps-finite-rhs simp add: eqs-subst-def finite-rhs-def*)

**lemma** *append-rhs-keeps-cl*:  
 $\text{classes-of } (\text{append-rhs-rexp } rhs \ r) = \text{classes-of } rhs$   
**apply** (*auto simp: classes-of-def append-rhs-rexp-def*)  
**apply** (*case-tac xa, auto simp: image-def*)  
**by** (*rule-tac x = SEQ ra r in exI, rule-tac x = Trn x ra in beXI, simp+*)

**lemma** *arden-variate-removes-cl*:  
 $\text{classes-of } (\text{arden-variate } Y \ yrhs) = \text{classes-of } yrhs - \{Y\}$   
**apply** (*simp add: arden-variate-def append-rhs-keeps-cls items-of-def*)  
**by** (*auto simp: classes-of-def*)

**lemma** *lefts-of-keeps-cl*:  
 $\text{lefts-of } (\text{eqs-subst } ES \ Y \ yrhs) = \text{lefts-of } ES$   
**by** (*auto simp: lefts-of-def eqs-subst-def*)

**lemma** *rhs-subst-updates-cl*:  
 $X \notin \text{classes-of } xrhs \implies$   
 $\text{classes-of } (\text{rhs-subst } rhs \ X \ xrhs) = \text{classes-of } rhs \cup \text{classes-of } xrhs - \{X\}$   
**apply** (*simp only: rhs-subst-def append-rhs-keeps-cl*  
 $\text{classes-of-union-distrib[THEN sym]}$ )  
**by** (*auto simp: classes-of-def items-of-def*)

**lemma** *eqs-subst-keeps-self-contained*:  
**fixes**  $Y$   
**assumes** *sc: self-contained*  $(ES \cup \{(Y, yrhs)\})$  (**is** *self-contained*  $?A$ )  
**shows** *self-contained*  $(\text{eqs-subst } ES \ Y \ (\text{arden-variate } Y \ yrhs))$   
(**is** *self-contained*  $?B$ )

**proof**–  
{ **fix**  $X \ xrhs'$   
**assume**  $(X, xrhs') \in ?B$   
**then obtain**  $xrhs$   
**where**  $xrhs-xrhs'$ :  $xrhs' = \text{rhs-subst } xrhs \ Y \ (\text{arden-variate } Y \ yrhs)$   
**and** *X-in*:  $(X, xrhs) \in ES$  **by** (*simp add: eqs-subst-def, blast*)  
**have**  $\text{classes-of } xrhs' \subseteq \text{lefts-of } ?B$   
**proof**–  
**have**  $\text{lefts-of } ?B = \text{lefts-of } ES$  **by** (*auto simp add: lefts-of-def eqs-subst-def*)  
**moreover have**  $\text{classes-of } xrhs' \subseteq \text{lefts-of } ES$   
**proof**–  
**have**  $\text{classes-of } xrhs' \subseteq$   
 $\text{classes-of } xrhs \cup \text{classes-of } (\text{arden-variate } Y \ yrhs) - \{Y\}$   
**proof**–



```

    have  $Y \notin \text{classes-of } (\text{arden-variate } Y \text{ yrhs})$ 
      using arden-variate-removes-cl by simp
    thus ?thesis using xrhs-xrhs' by (auto simp:rhs-subst-updates-cl)
  qed
  moreover have  $\text{classes-of } xrhs \subseteq \text{lefts-of } ES \cup \{Y\}$  using X-in sc
    apply (simp only:self-contained-def lefts-of-union-distrib[THEN sym])
    by (drule-tac x = (X, xrhs) in bspec, auto simp:lefts-of-def)
  moreover have  $\text{classes-of } (\text{arden-variate } Y \text{ yrhs}) \subseteq \text{lefts-of } ES \cup \{Y\}$ 
    using sc
    by (auto simp add:arden-variate-removes-cl self-contained-def lefts-of-def)
  ultimately show ?thesis by auto
  qed
  ultimately show ?thesis by simp
  qed
} thus ?thesis by (auto simp only:eqs-subst-def self-contained-def)
qed

lemma eqs-subst-satisfy-Inv:
  assumes Inv-ES:  $Inv (ES \cup \{(Y, yrhs)\})$ 
  shows  $Inv (\text{eqs-subst } ES \ Y \ (\text{arden-variate } Y \ \text{yrhs}))$ 
  proof -
    have finite-yrhs:  $finite \ yrhs$ 
      using Inv-ES by (auto simp:Inv-def finite-rhs-def)
    have nonempty-yrhs:  $rhs\ nonempty \ yrhs$ 
      using Inv-ES by (auto simp:Inv-def ardenable-def)
    have Y-eq-yrhs:  $Y = L \ yrhs$ 
      using Inv-ES by (simp only:Inv-def valid-egns-def, blast)
    have distinct-equas ( $\text{eqs-subst } ES \ Y \ (\text{arden-variate } Y \ \text{yrhs})$ )
      using Inv-ES
      by (auto simp:distinct-equas-def eqs-subst-def Inv-def)
    moreover have finite ( $\text{eqs-subst } ES \ Y \ (\text{arden-variate } Y \ \text{yrhs})$ )
      using Inv-ES by (simp add:Inv-def eqs-subst-keeps-finite)
    moreover have finite-rhs ( $\text{eqs-subst } ES \ Y \ (\text{arden-variate } Y \ \text{yrhs})$ )
  proof -
    have finite-rhs ES using Inv-ES
      by (simp add:Inv-def finite-rhs-def)
    moreover have finite ( $\text{arden-variate } Y \ \text{yrhs}$ )
  proof -
    have finite yrhs using Inv-ES
      by (auto simp:Inv-def finite-rhs-def)
    thus ?thesis using arden-variate-keeps-finite by simp
  qed
  ultimately show ?thesis
    by (simp add:eqs-subst-keeps-finite-rhs)
  qed
  moreover have ardenable ( $\text{eqs-subst } ES \ Y \ (\text{arden-variate } Y \ \text{yrhs})$ )
  proof -
    { fix X rhs
      assume  $(X, rhs) \in ES$ 

```

```

hence rhs-nonempty rhs using prems Inv-ES
  by (simp add:Inv-def ardenable-def)
with nonempty-yrhs
have rhs-nonempty (rhs-subst rhs Y (arden-variate Y yrhs))
  by (simp add:nonempty-yrhs
      rhs-subst-keeps-nonempty arden-variate-keeps-nonempty)
} thus ?thesis by (auto simp add:ardenable-def eqs-subst-def)
qed
moreover have valid-eqns (eqs-subst ES Y (arden-variate Y yrhs))
proof–
  have Y = L (arden-variate Y yrhs)
  using Y-eq-yrhs Inv-ES finite-yrhs nonempty-yrhs
  by (rule-tac arden-variate-keeps-eq, (simp add:rexp-of-empty)+)
thus ?thesis using Inv-ES
  by (clarsimp simp add:valid-eqns-def
      eqs-subst-def rhs-subst-keeps-eq Inv-def finite-rhs-def
      simp del:L-rhs.simps)
qed
moreover have
  non-empty-subst: non-empty (eqs-subst ES Y (arden-variate Y yrhs))
  using Inv-ES by (auto simp:Inv-def non-empty-def eqs-subst-def)
moreover
have self-subst: self-contained (eqs-subst ES Y (arden-variate Y yrhs))
  using Inv-ES eqs-subst-keeps-self-contained by (simp add:Inv-def)
ultimately show ?thesis using Inv-ES by (simp add:Inv-def)
qed

```

```

lemma eqs-subst-card-le:
  assumes finite: finite (ES::(string set × rhs-item set) set)
  shows card (eqs-subst ES Y yrhs) ≤ card ES
proof–
  def f ≡  $\lambda x. ((fst\ x)::string\ set, rhs-subst\ (snd\ x)\ Y\ yrhs)$ 
  have eqs-subst ES Y yrhs = f ‘ ES
  apply (auto simp:eqs-subst-def f-def image-def)
  by (rule-tac x = (Ya, yrhsa) in bexI, simp+)
  thus ?thesis using finite by (auto intro:card-image-le)
qed

```

```

lemma eqs-subst-cls-remains:
   $(X, xrhs) \in ES \implies \exists xrhs'. (X, xrhs') \in (eqs-subst\ ES\ Y\ yrhs)$ 
by (auto simp:eqs-subst-def)

```

```

lemma card-noteq-1-has-more:
  assumes card:card S ≠ 1
  and e-in: e ∈ S
  and finite: finite S
  obtains e' where e' ∈ S ∧ e ≠ e'
proof–
  have card (S - {e}) > 0

```

```

proof –
  have  $\text{card } S > 1$  using  $\text{card } e\text{-in } \text{finite}$ 
  by ( $\text{case-tac } \text{card } S, \text{auto}$ )
  thus  $?thesis$  using  $\text{finite } e\text{-in}$  by  $\text{auto}$ 
qed
hence  $S - \{e\} \neq \{\}$  using  $\text{finite}$  by ( $\text{rule-tac } \text{notI}, \text{simp}$ )
thus  $(\bigwedge e'. e' \in S \wedge e \neq e' \implies \text{thesis}) \implies \text{thesis}$  by  $\text{auto}$ 
qed

```

**lemma** *iteration-step*:

```

assumes  $\text{Inv-ES}: \text{Inv } ES$ 
and  $X\text{-in-ES}: (X, \text{xrhs}) \in ES$ 
and  $\text{not-T}: \text{card } ES \neq 1$ 
shows  $\exists ES'. (\text{Inv } ES' \wedge (\exists \text{xrhs}'. (X, \text{xrhs}') \in ES')) \wedge$ 
   $(\text{card } ES', \text{card } ES) \in \text{less-than}$  (is  $\exists ES'. ?P ES'$ )

```

**proof** –

```

have  $\text{finite-ES}: \text{finite } ES$  using  $\text{Inv-ES}$  by ( $\text{simp add:Inv-def}$ )
then obtain  $Y \text{ yrhs}$ 
  where  $Y\text{-in-ES}: (Y, \text{yrhs}) \in ES$  and  $\text{not-eq}: (X, \text{xrhs}) \neq (Y, \text{yrhs})$ 
  using  $\text{not-T } X\text{-in-ES}$  by ( $\text{drule-tac } \text{card-noteq-1-has-more}, \text{auto}$ )
def  $ES' == ES - \{(Y, \text{yrhs})\}$ 
let  $?ES'' = \text{eqs-subst } ES' Y$  ( $\text{arden-variate } Y \text{ yrhs}$ )
have  $?P ?ES''$ 
proof –
  have  $\text{Inv } ?ES''$  using  $Y\text{-in-ES } \text{Inv-ES}$ 
  by ( $\text{rule-tac } \text{eqs-subst-satisfy-Inv}, \text{simp add:ES'-def insert-absorb}$ )
moreover have  $\exists \text{xrhs}'. (X, \text{xrhs}') \in ?ES''$  using  $\text{not-eq } X\text{-in-ES}$ 
  by ( $\text{rule-tac } ES = ES'$  in  $\text{eqs-subst-cls-remains}, \text{auto simp add:ES'-def}$ )
moreover have  $(\text{card } ?ES'', \text{card } ES) \in \text{less-than}$ 
proof –
  have  $\text{finite } ES'$  using  $\text{finite-ES } ES'\text{-def}$  by  $\text{auto}$ 
moreover have  $\text{card } ES' < \text{card } ES$  using  $\text{finite-ES } Y\text{-in-ES}$ 
  by ( $\text{auto simp:ES'-def card-gt-0-iff intro:diff-Suc-less}$ )
ultimately show  $?thesis$ 
  by ( $\text{auto dest:eqs-subst-card-le elim:le-less-trans}$ )

```

**qed**

**ultimately show**  $?thesis$  **by**  $\text{simp}$

**qed**

**thus**  $?thesis$  **by**  $\text{blast}$

**qed**

#### 2.1.4 Conclusion of the proof

From this point until *hard-direction*, the hard direction is proved through a simple application of the iteration principle.

**lemma** *iteration-conc*:

```

assumes  $\text{history}: \text{Inv } ES$ 
and  $X\text{-in-ES}: \exists \text{xrhs}. (X, \text{xrhs}) \in ES$ 
shows

```

$\exists ES'. (Inv\ ES' \wedge (\exists\ xrhs'. (X, xrhs') \in ES')) \wedge card\ ES' = 1$   
(is  $\exists\ ES'. ?P\ ES'$ )

**proof** (cases card  $ES = 1$ )  
  **case** *True*  
  **thus** *?thesis using history X-in-ES*  
  **by** *blast*  
**next**  
  **case** *False*  
  **thus** *?thesis using history iteration-step X-in-ES*  
  **by** (rule-tac  $f = card$  in *wf-iter, auto*)  
**qed**

**lemma** *last-cl-exists-rexp*:  
  **assumes** *ES-single*:  $ES = \{(X, xrhs)\}$   
  **and** *Inv-ES*:  $Inv\ ES$   
  **shows**  $\exists (r::rexp). L\ r = X$  (is  $\exists\ r. ?P\ r$ )  
**proof** –  
  **let**  $?A = arden-variate\ X\ xrhs$   
  **have**  $?P (rexp-of-lam\ ?A)$   
  **proof** –  
  **have**  $L (rexp-of-lam\ ?A) = L (lam-of\ ?A)$   
  **proof**(rule *rexp-of-lam-eq-lam-set*)  
  **show** *finite (arden-variate X xrhs) using Inv-ES ES-single*  
  **by** (rule-tac *arden-variate-keeps-finite,*  
  *auto simp add:Inv-def finite-rhs-def*)  
**qed**  
  **also have**  $\dots = L\ ?A$   
  **proof**–  
  **have**  $lam-of\ ?A = ?A$   
  **proof**–  
  **have** *classes-of ?A = {} using Inv-ES ES-single*  
  **by** (*simp add:arden-variate-removes-cl*  
  *self-contained-def Inv-def lefts-of-def*)  
  **thus** *?thesis*  
  **by** (*auto simp only:lam-of-def classes-of-def, case-tac x, auto*)  
**qed**  
  **thus** *?thesis by simp*  
**qed**  
  **also have**  $\dots = X$   
  **proof**(rule *arden-variate-keeps-eq [THEN sym]*)  
  **show**  $X = L\ xrhs$  **using** *Inv-ES ES-single*  
  **by** (*auto simp only:Inv-def valid-eqns-def*)  
**next**  
  **from** *Inv-ES ES-single* **show**  $\square \notin L (rexp-of\ xrhs\ X)$   
  **by**(*simp add:Inv-def ardenable-def rexp-of-empty finite-rhs-def*)  
**next**  
  **from** *Inv-ES ES-single* **show** *finite xrhs*  
  **by** (*simp add:Inv-def finite-rhs-def*)  
**qed**

**finally show** *?thesis* **by** *simp*  
**qed**  
**thus** *?thesis* **by** *auto*  
**qed**

**lemma** *every-egcl-has-reg*:  
**assumes** *finite-CS*: *finite* (*UNIV* // ( $\approx$ *Lang*))  
**and** *X-in-CS*:  $X \in$  (*UNIV* // ( $\approx$ *Lang*))  
**shows**  $\exists$  (*reg*::*rexp*).  $L$  *reg* =  $X$  (**is**  $\exists$   $r$ . *?E*  $r$ )  
**proof** –  
**from** *X-in-CS* **have**  $\exists$  *xrhs*.  $(X, \textit{xrhs}) \in$  (*eqs* (*UNIV* // ( $\approx$ *Lang*)))  
**by** (*auto simp: eqs-def init-rhs-def*)  
**then obtain** *ES* *xrhs* **where** *Inv-ES*: *Inv ES*  
**and** *X-in-ES*:  $(X, \textit{xrhs}) \in$  *ES*  
**and** *card-ES*: *card ES* = 1  
**using** *finite-CS X-in-CS init-ES-satisfy-Inv iteration-conc*  
**by** *blast*  
**hence** *ES-single-equa*:  $ES = \{(X, \textit{xrhs})\}$   
**by** (*auto simp: Inv-def dest!: card-Suc-Diff1 simp: card-eq-0-iff*)  
**thus** *?thesis* **using** *Inv-ES*  
**by** (*rule last-cl-exists-rexp*)  
**qed**

**lemma** *finals-in-partitions*:  
*finals Lang*  $\subseteq$  (*UNIV* // ( $\approx$ *Lang*))  
**by** (*auto simp: finals-def quotient-def*)

**theorem** *hard-direction*:  
**assumes** *finite-CS*: *finite* (*UNIV* // ( $\approx$ *Lang*))  
**shows**  $\exists$  (*reg*::*rexp*).  $Lang = L$  *reg*  
**proof** –  
**have**  $\forall$   $X \in$  (*UNIV* // ( $\approx$ *Lang*)).  $\exists$  (*reg*::*rexp*).  $X = L$  *reg*  
**using** *finite-CS every-egcl-has-reg* **by** *blast*  
**then obtain**  $f$   
**where** *f-prop*:  $\forall$   $X \in$  (*UNIV* // ( $\approx$ *Lang*)).  $X = L$  ( $(f$   $X)::\textit{rexp}$ )  
**by** (*auto dest: bchoice*)  
**def** *rs*  $\equiv$   $f$  ‘ (*finals Lang*)  
**have**  $Lang = \bigcup$  (*finals Lang*) **using** *lang-is-union-of-finals* **by** *auto*  
**also have**  $\dots = L$  (*folds ALT NULL rs*)  
**proof** –  
**have** *finite rs*  
**proof** –  
**have** *finite* (*finals Lang*)  
**using** *finite-CS finals-in-partitions[of Lang]*  
**by** (*erule-tac finite-subset, simp*)  
**thus** *?thesis* **using** *rs-def* **by** *auto*  
**qed**  
**thus** *?thesis*  
**using** *f-prop rs-def finals-in-partitions[of Lang]* **by** *auto*

qed  
 finally show *?thesis* by *blast*  
 qed

### 3 Direction: *regular language* $\Rightarrow$ *finite partition*

#### 3.1 The scheme for this direction

The following convenient notation  $x \approx_{Lang} y$  means: string  $x$  and  $y$  are equivalent with respect to language  $Lang$ .

**definition**

*str-eq* (-  $\approx$  -)

**where**

$x \approx_{Lang} y \equiv (x, y) \in (\approx_{Lang})$

The very basic scheme to show the finiteness of the partition generated by a language  $Lang$  is by attaching tags to every string. The set of tags are carefully chosen to make it finite. If it can be proved that strings with the same tag are equivalent with respect  $Lang$ , then the partition given rise by  $Lang$  must be finite. The reason for this is a lemma in standard library (*finite-imageD*), which says: if the image of an injective function on a set  $A$  is finite, then  $A$  is finite. It can be shown that the function obtained by lifting  $tag$  to the level of equivalent classes (i.e.  $(op \text{ ' } tag)$ ) is injective (by lemma *tag-image-injI*) and the image of this function is finite (with the help of lemma *finite-tag-imageI*). This argument is formalized by the following lemma *tag-finite-imageD*.

**lemma** *tag-finite-imageD*:

**assumes** *str-inj*:  $\bigwedge m n. tag\ m = tag\ (n::string) \implies m \approx_{lang} n$

**and** *range*: *finite* (range *tag*)

**shows** *finite* (*UNIV* //  $(\approx_{lang})$ )

**proof** (*rule-tac*  $f = (op \text{ ' } tag)$  **in** *finite-imageD*)

**show** *finite*  $(op \text{ ' } tag \text{ ' } UNIV // \approx_{lang})$  **using** *range*

**apply** (*rule-tac*  $B = Pow\ (tag \text{ ' } UNIV)$  **in** *finite-subset*)

**by** (*auto simp add:image-def Pow-def*)

**next**

**show** *inj-on*  $(op \text{ ' } tag)$  (*UNIV* //  $\approx_{lang}$ )

**proof**–

{ **fix**  $X\ Y$

**assume** *X-in*:  $X \in UNIV // \approx_{lang}$

**and** *Y-in*:  $Y \in UNIV // \approx_{lang}$

**and** *tag-eq*:  $tag \text{ ' } X = tag \text{ ' } Y$

**then obtain**  $x\ y$  **where**  $x \in X$  **and**  $y \in Y$  **and**  $tag\ x = tag\ y$

**unfolding** *quotient-def Image-def str-eq-rel-def str-eq-def image-def*

**apply** *simp* **by** *blast*

**with** *X-in Y-in str-inj*[*of*  $x\ y$ ]

**have**  $X = Y$  **by** (*auto simp:quotient-def str-eq-rel-def str-eq-def*)

} **thus** *?thesis* **unfolding** *inj-on-def* **by** *auto*

qed  
qed

### 3.2 Lemmas for basic cases

The the final result of this direction is in *easier-direction*, which is an induction on the structure of regular expressions. There is one case for each regular expression operator. For basic operators such as *NULL*, *EMPTY*, *CHAR*  $c$ , the finiteness of their language partition can be established directly with no need of tagging. This section contains several technical lemma for these base cases.

The inductive cases involve operators *ALT*, *SEQ* and *STAR*. Tagging functions need to be defined individually for each of them. There will be one dedicated section for each of these cases, and each section goes virtually the same way: gives definition of the tagging function and prove that strings with the same tag are equivalent.

**lemma** *quot-empty-subset*:

$UNIV // (\approx\{\emptyset\}) \subseteq \{\{\emptyset\}, UNIV - \{\emptyset\}\}$

**proof**

**fix**  $x$

**assume**  $x \in UNIV // \approx\{\emptyset\}$

**then obtain**  $y$  **where**  $h: x = \{z. (y, z) \in \approx\{\emptyset\}\}$

**unfolding** *quotient-def Image-def* **by** *blast*

**show**  $x \in \{\{\emptyset\}, UNIV - \{\emptyset\}\}$

**proof** (*cases*  $y = \emptyset$ )

**case** *True* **with**  $h$

**have**  $x = \{\emptyset\}$  **by** (*auto simp:str-eq-rel-def*)

**thus** *?thesis* **by** *simp*

**next**

**case** *False* **with**  $h$

**have**  $x = UNIV - \{\emptyset\}$  **by** (*auto simp:str-eq-rel-def*)

**thus** *?thesis* **by** *simp*

qed

qed

**lemma** *quot-char-subset*:

$UNIV // (\approx\{[c]\}) \subseteq \{\{\emptyset\}, \{[c]\}, UNIV - \{\emptyset, [c]\}\}$

**proof**

**fix**  $x$

**assume**  $x \in UNIV // \approx\{[c]\}$

**then obtain**  $y$  **where**  $h: x = \{z. (y, z) \in \approx\{[c]\}\}$

**unfolding** *quotient-def Image-def* **by** *blast*

**show**  $x \in \{\{\emptyset\}, \{[c]\}, UNIV - \{\emptyset, [c]\}\}$

**proof** –

{ **assume**  $y = \emptyset$  **hence**  $x = \{\emptyset\}$  **using**  $h$

**by** (*auto simp:str-eq-rel-def*)

} **moreover** {

**assume**  $y = [c]$  **hence**  $x = \{[c]\}$  **using**  $h$   
**by** (*auto dest!:spec[where  $x = []$ ] simp:str-eq-rel-def*)  
**}** **moreover** {  
**assume**  $y \neq []$  **and**  $y \neq [c]$   
**hence**  $\forall z. (y @ z) \neq [c]$  **by** (*case-tac y, auto*)  
**moreover have**  $\bigwedge p. (p \neq [] \wedge p \neq [c]) = (\forall q. p @ q \neq [c])$   
**by** (*case-tac p, auto*)  
**ultimately have**  $x = UNIV - \{[], [c]\}$  **using**  $h$   
**by** (*auto simp add:str-eq-rel-def*)  
**}** **ultimately show** *?thesis* **by** *blast*  
**qed**  
**qed**

### 3.3 The case for SEQ

**definition**

*tag-str-SEQ*  $L_1 L_2 x \equiv$   
 $((\approx L_1) \text{ “ } \{x\}, \{(\approx L_2) \text{ “ } \{x - xa\} \mid xa. xa \leq x \wedge xa \in L_1\})$

**lemma** *tag-str-seq-range-finite*:

$\llbracket \text{finite } (UNIV // \approx L_1); \text{finite } (UNIV // \approx L_2) \rrbracket$   
 $\implies \text{finite } (\text{range } (\text{tag-str-SEQ } L_1 L_2))$

**apply** (*rule-tac B = (UNIV //  $\approx L_1$ )  $\times$  (Pow (UNIV //  $\approx L_2$ )) in finite-subset*)  
**by** (*auto simp:tag-str-SEQ-def Image-def quotient-def split:if-splits*)

**lemma** *append-seq-elim*:

**assumes**  $x @ y \in L_1 ;; L_2$   
**shows**  $(\exists xa \leq x. xa \in L_1 \wedge (x - xa) @ y \in L_2) \vee$   
 $(\exists ya \leq y. (x @ ya) \in L_1 \wedge (y - ya) \in L_2)$

**proof**–

**from** *assms* **obtain**  $s_1 s_2$   
**where**  $x @ y = s_1 @ s_2$   
**and** *in-seq*:  $s_1 \in L_1 \wedge s_2 \in L_2$   
**by** (*auto simp:Seq-def*)  
**hence**  $(x \leq s_1 \wedge (s_1 - x) @ s_2 = y) \vee (s_1 \leq x \wedge (x - s_1) @ y = s_2)$   
**using** *app-eq-dest* **by** *auto*  
**moreover have**  $\llbracket x \leq s_1; (s_1 - x) @ s_2 = y \rrbracket \implies$   
 $\exists ya \leq y. (x @ ya) \in L_1 \wedge (y - ya) \in L_2$   
**using** *in-seq* **by** (*rule-tac x = s\_1 - x in exI, auto elim:prefixE*)  
**moreover have**  $\llbracket s_1 \leq x; (x - s_1) @ y = s_2 \rrbracket \implies$   
 $\exists xa \leq x. xa \in L_1 \wedge (x - xa) @ y \in L_2$   
**using** *in-seq* **by** (*rule-tac x = s\_1 in exI, auto*)  
**ultimately show** *?thesis* **by** *blast*

**qed**

**lemma** *tag-str-SEQ-injI*:

*tag-str-SEQ*  $L_1 L_2 m = \text{tag-str-SEQ } L_1 L_2 n \implies m \approx (L_1 ;; L_2) n$

**proof**–

{ **fix**  $x y z$



**assume**  $xz\text{-in-seq}: x @ z \in L_1 ;; L_2$   
**and**  $tag\text{-}xy: tag\text{-}str\text{-}SEQ L_1 L_2 x = tag\text{-}str\text{-}SEQ L_1 L_2 y$   
**have**  $y @ z \in L_1 ;; L_2$   
**proof**–  
**have**  $(\exists xa \leq x. xa \in L_1 \wedge (x - xa) @ z \in L_2) \vee$   
 $(\exists za \leq z. (x @ za) \in L_1 \wedge (z - za) \in L_2)$   
**using**  $xz\text{-in-seq}$  *append-seq-elim* **by** *simp*  
**moreover** {  
**fix**  $xa$   
**assume**  $h1: xa \leq x$  **and**  $h2: xa \in L_1$  **and**  $h3: (x - xa) @ z \in L_2$   
**obtain**  $ya$  **where**  $ya \leq y$  **and**  $ya \in L_1$  **and**  $(y - ya) @ z \in L_2$   
**proof** –  
**have**  $\exists ya. ya \leq y \wedge ya \in L_1 \wedge (x - xa) \approx_{L_2} (y - ya)$   
**proof** –  
**have**  $\{\approx_{L_2} \text{ “ } \{x - xa\} | xa. xa \leq x \wedge xa \in L_1 \} =$   
 $\{\approx_{L_2} \text{ “ } \{y - xa\} | xa. xa \leq y \wedge xa \in L_1 \}$   
 $(\text{is } ?Left = ?Right)$   
**using**  $h1$   $tag\text{-}xy$  **by**  $(auto\ simp:tag\text{-}str\text{-}SEQ\text{-}def)$   
**moreover** **have**  $\approx_{L_2} \text{ “ } \{x - xa\} \in ?Left$  **using**  $h1$   $h2$  **by** *auto*  
**ultimately** **have**  $\approx_{L_2} \text{ “ } \{x - xa\} \in ?Right$  **by** *simp*  
**thus**  $?thesis$  **by**  $(auto\ simp:Image\text{-}def\ str\text{-}eq\text{-}rel\text{-}def\ str\text{-}eq\text{-}def)$   
**qed**  
**with** *prems* **show**  $?thesis$  **by**  $(auto\ simp:str\text{-}eq\text{-}rel\text{-}def\ str\text{-}eq\text{-}def)$   
**qed**  
**hence**  $y @ z \in L_1 ;; L_2$  **by**  $(erule\text{-}tac\ prefixE, auto\ simp:Seq\text{-}def)$   
**}** **moreover** {  
**fix**  $za$   
**assume**  $h1: za \leq z$  **and**  $h2: (x @ za) \in L_1$  **and**  $h3: z - za \in L_2$   
**hence**  $y @ za \in L_1$   
**proof**–  
**have**  $\approx_{L_1} \text{ “ } \{x\} = \approx_{L_1} \text{ “ } \{y\}$   
**using**  $h1$   $tag\text{-}xy$  **by**  $(auto\ simp:tag\text{-}str\text{-}SEQ\text{-}def)$   
**with**  $h2$  **show**  $?thesis$   
**by**  $(auto\ simp:Image\text{-}def\ str\text{-}eq\text{-}rel\text{-}def\ str\text{-}eq\text{-}def)$   
**qed**  
**with**  $h1$   $h3$  **have**  $y @ z \in L_1 ;; L_2$   
**by**  $(drule\text{-}tac\ A = L_1$  **in**  $seq\text{-}intro, auto\ elim:prefixE)$   
**}**  
**ultimately** **show**  $?thesis$  **by** *blast*  
**qed**  
**}** **thus**  $tag\text{-}str\text{-}SEQ L_1 L_2 m = tag\text{-}str\text{-}SEQ L_1 L_2 n \implies m \approx_{(L_1 ;; L_2)} n$   
**by**  $(auto\ simp\ add: str\text{-}eq\text{-}def\ str\text{-}eq\text{-}rel\text{-}def)$   
**qed**

**lemma** *quot-seq-finiteI*:

$\llbracket finite (UNIV // \approx_{L_1}); finite (UNIV // \approx_{L_2}) \rrbracket$

$\implies finite (UNIV // \approx_{(L_1 ;; L_2)})$

**apply**  $(rule\text{-}tac\ tag = tag\text{-}str\text{-}SEQ L_1 L_2$  **in**  $tag\text{-}finite\text{-}imageD)$

**by**  $(auto\ intro:tag\text{-}str\text{-}SEQ\text{-}injI\ elim:tag\text{-}str\text{-}seq\text{-}range\text{-}finite)$

### 3.4 The case for *ALT*

**definition**

$tag\text{-}str\text{-}ALT\ L_1\ L_2\ (x::string) \equiv ((\approx L_1) \text{ `` } \{x\}, (\approx L_2) \text{ `` } \{x\})$

**lemma** *quot-union-finiteI*:

**assumes** *finite1*:  $finite\ (UNIV\ //\ \approx(L_1::string\ set))$

**and** *finite2*:  $finite\ (UNIV\ //\ \approx L_2)$

**shows**  $finite\ (UNIV\ //\ \approx(L_1\ \cup\ L_2))$

**proof** (*rule-tac tag = tag-str-ALT L1 L2 in tag-finite-imageD*)

**show**  $\bigwedge m\ n.\ tag\text{-}str\text{-}ALT\ L_1\ L_2\ m = tag\text{-}str\text{-}ALT\ L_1\ L_2\ n \implies m \approx(L_1\ \cup\ L_2)\ n$

**unfolding** *tag-str-ALT-def str-eq-def Image-def str-eq-rel-def* **by** *auto*

**next**

**show**  $finite\ (range\ (tag\text{-}str\text{-}ALT\ L_1\ L_2))$  **using** *finite1 finite2*

**apply** (*rule-tac B = (UNIV // ≈L1) × (UNIV // ≈L2) in finite-subset*)

**by** (*auto simp:tag-str-ALT-def Image-def quotient-def*)

**qed**

### 3.5 The case for *STAR*

This turned out to be the trickiest case.

**definition**

$tag\text{-}str\text{-}STAR\ L_1\ x \equiv \{(\approx L_1) \text{ `` } \{x - xa\} \mid xa.\ xa < x \wedge xa \in L_1^*\}$

**lemma** *finite-set-has-max*:  $\llbracket finite\ A; A \neq \{\} \rrbracket \implies$

$(\exists\ max \in A.\ \forall\ a \in A.\ f\ a \leq (f\ max :: nat))$

**proof** (*induct rule:finite.induct*)

**case** *emptyI* **thus** *?case* **by** *simp*

**next**

**case** (*insertI A a*)

**show** *?case*

**proof** (*cases A = {}*)

**case** *True* **thus** *?thesis* **by** (*rule-tac x = a in beXI, auto*)

**next**

**case** *False*

**with** *prems* **obtain** *max*

**where** *h1*:  $max \in A$

**and** *h2*:  $\forall a \in A.\ f\ a \leq f\ max$  **by** *blast*

**show** *?thesis*

**proof** (*cases f a ≤ f max*)

**assume**  $f\ a \leq f\ max$

**with** *h1 h2* **show** *?thesis* **by** (*rule-tac x = max in beXI, auto*)

**next**

**assume**  $\neg (f\ a \leq f\ max)$

**thus** *?thesis* **using** *h2* **by** (*rule-tac x = a in beXI, auto*)

**qed**

**qed**

**qed**

**lemma** *finite-strict-prefix-set*:  $finite \{xa. xa < (x::string)\}$   
**apply** (*induct*  $x$  *rule:rev-induct, simp*)  
**apply** (*subgoal-tac*  $\{xa. xa < xs @ [x]\} = \{xa. xa < xs\} \cup \{xs\}$ )  
**by** (*auto simp:strict-prefix-def*)

**lemma** *tag-str-star-range-finite*:  
 $finite (UNIV // \approx_{L_1}) \implies finite (range (tag-str-STAR L_1))$   
**apply** (*rule-tac*  $B = Pow (UNIV // \approx_{L_1})$  **in** *finite-subset*)  
**by** (*auto simp:tag-str-STAR-def Image-def*  
*quotient-def split:if-splits*)

**lemma** *tag-str-STAR-injI*:  
 $tag-str-STAR L_1 m = tag-str-STAR L_1 n \implies m \approx_{(L_1^*)} n$

**proof** –

{ **fix**  $x y z$   
**assume**  $xz-in-star: x @ z \in L_1^*$   
**and**  $tag-xy: tag-str-STAR L_1 x = tag-str-STAR L_1 y$   
**have**  $y @ z \in L_1^*$   
**proof**(*cases*  $x = []$ )  
**case** *True*  
**with**  $tag-xy$  **have**  $y = []$   
**by** (*auto simp:tag-str-STAR-def strict-prefix-def*)  
**thus** *?thesis* **using**  $xz-in-star$  *True* **by** *simp*

**next**  
**case** *False*  
**obtain**  $x-max$   
**where**  $h1: x-max < x$   
**and**  $h2: x-max \in L_1^*$   
**and**  $h3: (x - x-max) @ z \in L_1^*$   
**and**  $h4: \forall xa < x. xa \in L_1^* \wedge (x - xa) @ z \in L_1^* \implies length xa \leq length x-max$

**proof** –

**let**  $?S = \{xa. xa < x \wedge xa \in L_1^* \wedge (x - xa) @ z \in L_1^*\}$   
**have** *finite*  $?S$   
**by** (*rule-tac*  $B = \{xa. xa < x\}$  **in** *finite-subset*,  
*auto simp:finite-strict-prefix-set*)  
**moreover** **have**  $?S \neq \{\}$  **using** *False xz-in-star*  
**by** (*simp, rule-tac*  $x = []$  **in** *exI, auto simp:strict-prefix-def*)  
**ultimately** **have**  $\exists max \in ?S. \forall a \in ?S. length a \leq length max$   
**using** *finite-set-has-max* **by** *blast*  
**with** *prems* **show** *?thesis* **by** *blast*

**qed**

**obtain**  $ya$

**where**  $h5: ya < y$  **and**  $h6: ya \in L_1^*$  **and**  $h7: (x - x-max) \approx_{L_1} (y - ya)$

**proof** –

**from**  $tag-xy$  **have**  $\{\approx_{L_1} \{x - xa\} \mid xa. xa < x \wedge xa \in L_1^*\} =$   
 $\{\approx_{L_1} \{y - xa\} \mid xa. xa < y \wedge xa \in L_1^*\}$  (**is** *?left = ?right*)  
**by** (*auto simp:tag-str-STAR-def*)

**moreover have**  $\approx_{L_1}$  “  $\{x - x\text{-max}\} \in ?\text{left}$  **using**  $h1$   $h2$  **by** *auto*  
**ultimately have**  $\approx_{L_1}$  “  $\{x - x\text{-max}\} \in ?\text{right}$  **by** *simp*  
**with** *prems* **show** *?thesis* **apply**  
*(simp add:Image-def str-eq-rel-def str-eq-def)* **by** *blast*  
**qed**  
**have**  $(y - ya) @ z \in L_1^\star$   
**proof**–  
**from**  $h3$   $h1$  **obtain**  $a$   $b$  **where** *a-in*:  $a \in L_1$   
**and** *a-neq*:  $a \neq []$  **and** *b-in*:  $b \in L_1^\star$   
**and** *ab-max*:  $(x - x\text{-max}) @ z = a @ b$   
**by** (*drule-tac star-decom*, *auto simp:strict-prefix-def elim:prefixE*)  
**have**  $(x - x\text{-max}) \leq a \wedge (a - (x - x\text{-max})) @ b = z$   
**proof** –  
**have**  $((x - x\text{-max}) \leq a \wedge (a - (x - x\text{-max})) @ b = z) \vee$   
 $(a < (x - x\text{-max}) \wedge ((x - x\text{-max}) - a) @ z = b)$   
**using** *app-eq-dest[OF ab-max]* **by** (*auto simp:strict-prefix-def*)  
**moreover** {  
**assume** *np*:  $a < (x - x\text{-max})$  **and** *b-eqs*:  $((x - x\text{-max}) - a) @ z = b$   
**have** *False*  
**proof** –  
**let**  $?x\text{-max}' = x\text{-max} @ a$   
**have**  $?x\text{-max}' < x$   
**using** *np h1* **by** (*clarsimp simp:strict-prefix-def diff-prefix*)  
**moreover have**  $?x\text{-max}' \in L_1^\star$   
**using** *a-in h2* **by** (*simp add:star-intro3*)  
**moreover have**  $(x - ?x\text{-max}') @ z \in L_1^\star$   
**using** *b-eqs b-in np h1* **by** (*simp add:diff-diff-appd*)  
**moreover have**  $\neg (\text{length } ?x\text{-max}' \leq \text{length } x\text{-max})$   
**using** *a-neq* **by** *simp*  
**ultimately show** *?thesis* **using**  $h4$  **by** *blast*  
**qed**  
**}** **ultimately show** *?thesis* **by** *blast*  
**qed**  
**then obtain**  $za$  **where** *z-decom*:  $z = za @ b$   
**and** *x-za*:  $(x - x\text{-max}) @ za \in L_1$   
**using** *a-in* **by** (*auto elim:prefixE*)  
**from** *x-za h7* **have**  $(y - ya) @ za \in L_1$   
**by** (*auto simp:str-eq-def str-eq-rel-def*)  
**with** *z-decom b-in* **show** *?thesis* **by** (*auto dest!:step[of (y - ya) @ za]*)  
**qed**  
**with**  $h5$   $h6$  **show** *?thesis*  
**by** (*drule-tac star-intro1*, *auto simp:strict-prefix-def elim:prefixE*)  
**qed**  
**}** **thus** *tag-str-STAR*  $L_1$   $m = \text{tag-str-STAR } L_1$   $n \implies m \approx_{(L_1^\star)} n$   
**by** (*auto simp add:str-eq-def str-eq-rel-def*)  
**qed**

**lemma** *quot-star-finiteI*:

*finite (UNIV //  $\approx_{L_1}$ )  $\implies$  finite (UNIV //  $\approx_{(L_1^\star)}$ )*

**apply** (*rule-tac tag = tag-str-STAR L<sub>1</sub> in tag-finite-imageD*)  
**by** (*auto intro:tag-str-STAR-injI elim:tag-str-star-range-finite*)

### 3.6 The main lemma

**lemma** *easier-directionv*:

*Lang = L (r::rexp)  $\implies$  finite (UNIV // ( $\approx$ Lang))*

**proof** (*induct arbitrary:Lang rule:rexp.induct*)

**case** *NULL*

**have** *UNIV // ( $\approx$ {})  $\subseteq$  {UNIV}*

**by** (*auto simp:quotient-def str-eq-rel-def str-eq-def*)

**with** *prems show ?case by (auto intro:finite-subset)*

**next**

**case** *EMPTY*

**have** *UNIV // ( $\approx$ {[]})  $\subseteq$  {{[]}, UNIV - {[]}}*

**by** (*rule quot-empty-subset*)

**with** *prems show ?case by (auto intro:finite-subset)*

**next**

**case** (*CHAR c*)

**have** *UNIV // ( $\approx$ {[c]})  $\subseteq$  {[[]],[c]}, UNIV - {[], [c]}*

**by** (*rule quot-char-subset*)

**with** *prems show ?case by (auto intro:finite-subset)*

**next**

**case** (*SEQ r<sub>1</sub> r<sub>2</sub>*)

**have**  $\llbracket$ *finite (UNIV //  $\approx$ (L r<sub>1</sub>)); finite (UNIV //  $\approx$ (L r<sub>2</sub>))* $\rrbracket$   
 $\implies$  *finite (UNIV //  $\approx$ (L r<sub>1</sub> ;; L r<sub>2</sub>))*

**by** (*erule quot-seq-finiteI, simp*)

**with** *prems show ?case by simp*

**next**

**case** (*ALT r<sub>1</sub> r<sub>2</sub>*)

**have**  $\llbracket$ *finite (UNIV //  $\approx$ (L r<sub>1</sub>)); finite (UNIV //  $\approx$ (L r<sub>2</sub>))* $\rrbracket$   
 $\implies$  *finite (UNIV //  $\approx$ (L r<sub>1</sub>  $\cup$  L r<sub>2</sub>))*

**by** (*erule quot-union-finiteI, simp*)

**with** *prems show ?case by simp*

**next**

**case** (*STAR r*)

**have** *finite (UNIV //  $\approx$ (L r))*  
 $\implies$  *finite (UNIV //  $\approx$ ((L r) $\star$ ))*

**by** (*erule quot-star-finiteI*)

**with** *prems show ?case by simp*

**qed**

**end**