

tphols-2011

By xingyuan

January 26, 2011

Contents

```
theory Prefix-subtract
imports Main List-Prefix
begin
```

1 A small theory of prefix subtraction

The notion of *prefix-subtract* is need to make proofs more readable.

```
fun prefix-subtract :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list (infix - 51)
```

```
where
```

```
  prefix-subtract [] xs = []
| prefix-subtract (x#xs) [] = x#xs
| prefix-subtract (x#xs) (y#ys) = (if x = y then prefix-subtract xs ys else (x#xs))
```

```
lemma [simp]: (x @ y) - x = y
```

```
apply (induct x)
```

```
by (case-tac y, simp+)
```

```
lemma [simp]: x - x = []
```

```
by (induct x, auto)
```

```
lemma [simp]: x = xa @ y  $\implies$  x - xa = y
```

```
by (induct x, auto)
```

```
lemma [simp]: x - [] = x
```

```
by (induct x, auto)
```

```
lemma [simp]: (x - y = [])  $\implies$  (x  $\leq$  y)
```

```
proof -
```

```
  have  $\exists xa. x = xa @ (x - y) \wedge xa \leq y$ 
```

```
    apply (rule prefix-subtract.induct[of - x y], simp+)
```

```
    by (clarsimp, rule-tac x = y # xa in exI, simp+)
```

```
  thus (x - y = [])  $\implies$  (x  $\leq$  y) by simp
```

```
qed
```

lemma *diff-prefix*:

$\llbracket c \leq a - b; b \leq a \rrbracket \implies b @ c \leq a$
by (*auto elim:prefixE*)

lemma *diff-diff-appd*:

$\llbracket c < a - b; b < a \rrbracket \implies (a - b) - c = a - (b @ c)$
apply (*clarsimp simp:strict-prefix-def*)
by (*drule diff-prefix, auto elim:prefixE*)

lemma *app-eq-cases*[*rule-format*]:

$\forall x . x @ y = m @ n \longrightarrow (x \leq m \vee m \leq x)$
apply (*induct y, simp*)
apply (*clarify, drule-tac x = x @ [a] in spec*)
by (*clarsimp, auto simp:prefix-def*)

lemma *app-eq-dest*:

$x @ y = m @ n \implies$
 $(x \leq m \wedge (m - x) @ n = y) \vee (m \leq x \wedge (x - m) @ y = n)$
by (*frule-tac app-eq-cases, auto elim:prefixE*)

end

theory *Myhill*

imports *Main List-Prefix Prefix-subtract*
begin

2 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*]: $\square \in L^*$
step[*intro*]: $\llbracket s1 \in L; s2 \in L^* \rrbracket \implies s1@s2 \in L^*$

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*:

$\llbracket x \in A; y \in B \rrbracket \implies x @ y \in A ;; B$

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*]: $x \in \text{lang}\star \implies \forall y. y \in \text{lang}\star \longrightarrow x @ y \in \text{lang}\star$

by (*erule Star.induct, auto*)

lemma *star-intro2*: $y \in \text{lang} \implies y \in \text{lang}\star$

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

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

$x \in \text{lang}\star \implies \forall y. y \in \text{lang} \longrightarrow x @ y \in \text{lang}\star$

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

lemma *star-decom*:

$\llbracket x \in \text{lang}\star; x \neq [] \rrbracket \implies (\exists a b. x = a @ b \wedge a \neq [] \wedge a \in \text{lang} \wedge b \in \text{lang}\star)$

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

lemma *star-decom'*:

$\llbracket x \in \text{lang}\star; x \neq [] \rrbracket \implies \exists a b. x = a @ b \wedge a \in \text{lang}\star \wedge b \in \text{lang}$

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'*)

then have $B ;; A\star = B ;; (\{\} \cup A\star ;; A)$

unfolding *Seq-def* **by** *simp*

also have $\dots = B \cup B ;; (A\star ;; A)$

unfolding *Seq-def* **by** *auto*

also have $\dots = B \cup (B ;; A\star) ;; A$

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  $X = B ;; A^*$ 
proof
  show  $B ;; A^* \subseteq X$ 
  proof-
    { fix  $x y$ 
      have  $[y \in A^*; x \in X] \implies x @ y \in X$ 
        apply (induct arbitrary:x rule:Star.induct, simp)
        by (auto simp only:append-assoc[THEN sym] dest:c2')
      } thus ?thesis using  $c1'$  by (auto simp:Seq-def)
  qed
next
show  $X \subseteq B ;; A^*$ 
proof-
  { fix  $x$ 
    have  $x \in X \implies x \in B ;; A^*$ 
    proof (induct x taking:length rule:measure-induct)
      fix  $z$ 
      assume hyps:
         $\forall y. \text{length } y < \text{length } z \longrightarrow y \in X \longrightarrow y \in B ;; A^*$ 
        and z-in:  $z \in X$ 
      show  $z \in B ;; A^*$ 
      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  $z_a z_b$  where z_a-in:  $z_a \in X$ 
            and  $z_a b: z = z_a @ z_b \wedge z_b \in A$  and z_bne:  $z_b \neq []$ 
            using nemp unfolding Seq-def by blast
          from z_bne z_a b have  $\text{length } z_a < \text{length } z$  by auto
          with z_a-in hyps have  $z_a \in B ;; A^*$  by blast
          hence  $z_a @ z_b \in B ;; A^*$  using  $z_a b$ 
            by (clarsimp simp:Seq-def, blast dest:star-intro3)
          thus ?thesis using  $z_a b$  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 $L(rexp)$ 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 $fold$ s is defined. The use of $SOME$ makes $fold$ more robust than the $fold$ in Isabelle library. The expression $fold$ s f makes sense when f is not *associative* and *commutitive*, while $fold\ f$ does not.

definition

$fold$ s $:: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set \Rightarrow 'b$

where

$fold$ s $f\ z\ S \equiv SOME\ x.\ fold-graph\ f\ z\ S\ x$

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

lemma $fold$ s-alt-simp [simp]:

$finite\ rs \Longrightarrow L\ (fold$ s $ALT\ NULL\ rs) = \bigcup (L\ `\ rs)$

apply (rule set-ext, simp add:fold-s-def)

apply (rule someI2-ex, erule finite-imp-fold-graph)

by (erule fold-graph.induct, auto)

lemma [simp]:

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

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

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

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

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

lemma *lang-is-union-of-finals*:

$$Lang = \bigcup finals(Lang)$$

proof

$$\text{show } Lang \subseteq \bigcup (finals Lang)$$

proof

fix x

assume $x \in Lang$

thus $x \in \bigcup (finals Lang)$

apply (*simp add:finals-def, rule-tac x = ($\approx Lang$) “ $\{x\}$ in exI*)

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

qed

next

$$\text{show } \bigcup (finals Lang) \subseteq Lang$$

apply (*clarsimp simp:finals-def str-eq-rel-def*)

by (*drule-tac x = [] in spec, auto*)

qed

3 Direction *finite partition* \Rightarrow *regular language*

The relationship between equivalent classes can be described by an equational system. For example, in equational system (??), 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 λ). 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 data type *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 λ in (??), 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 λ 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-rhs rhs =  $\bigcup$  (L ' rhs)
end
```

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

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

In the definition of $init-rhs$, the term $\{Trn Y (CHAR c) | Y c. Y \in CS \wedge Y ;; \{[c]\} \subseteq X\}$ 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 λ in (??).

With the help of $init-rhs$, 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-rhs CS X) | X. X  $\in$  CS}
```

The following *items-of rhs X* returns all *X*-items in *rhs*.

definition

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

The following *rexp-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-variate* and *rhs-subst*.

definition

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

The following *lam-of rhs* returns all pure regular expression items in *rhs*.

definition

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

The following *rexp-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-of-lam rhs* is used to compute the regular expression corresponds to *rhs*.

definition

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

The following *attach-rexp rexp' itm* attach the regular expression *rexp'* to the right of right hand side item *itm*.

fun *attach-rexp* :: *rexp* ⇒ *rhs-item* ⇒ *rhs-item*

where

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

The following *append-rhs-rexp rhs rexp* attaches *rexp* to every item in *rhs*.

definition

$$\text{append-rhs-rexp } \text{rhs } \text{rexp} \equiv (\text{attach-rexp } \text{rexp}) \text{ ' rhs}$$

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

definition

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

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

definition

$$\begin{aligned} & \text{rhs-subst rhs } X \text{ } xrhs \equiv \\ & (rhs - (\text{items-of rhs } X)) \cup (\text{append-rhs-rexp } xrhs \text{ } (\text{rexp-of rhs } X)) \end{aligned}$$

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

definition

$$eqs\text{-subst } ES \ X \ xrhs \equiv \{(Y, \text{rhs-subst } yrhs \ X \ xrhs) \mid Y \text{ } yrhs. (Y, yrhs) \in ES\}$$

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*: $(f \ e', f \ x) \in \text{less-than}$ **by** *auto*

show *?thesis*

proof(*rule h*)

from *ltf* **show** $(e', x) \in \text{inv-image less-than } f$

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

distinct-equas $ES \equiv$

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

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

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 *rhs* does not contain empty string. This is necessary for the application of Arden's transformation to *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 *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 *rhs*.

definition

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

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

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

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

definition

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

3.1 The proof of this direction

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

proof –

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

have *?B* = *(snd o the-Trn)* ‘*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!*]:

$P (\text{Trn } X r) \implies (\exists a. (\exists r. a = \text{Trn } X r \wedge P a))$ **by** *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 $L (\text{items-of rhs } X) = X ;; (L (\text{rexp-of rhs } X))$

proof –

have *finite* $((\text{snd} \circ \text{the-Trn})$ ‘*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 $L (\text{rexp-of-lam rhs}) = L (\text{lam-of rhs})$

proof –

have *finite* $(\text{the-r}$ ‘ $\{\text{Lam } r \mid r. \text{Lam } r \in \text{rhs}\}$) **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]*:
 $L (\text{attach-rop } r \text{ } xb) = L \text{ } xb ;; L \text{ } r$
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*:
 $L (\text{append-rhs-rop } rhs \text{ } r) = L \text{ } rhs ;; L \text{ } r$
apply (*auto simp:append-rhs-rop-def image-def*)
apply (*auto simp:Seq-def*)
apply (*rule-tac* $x = L \text{ } xb ;; L \text{ } r$ **in** exI , *auto simp add:Seq-def*)
by (*rule-tac* $x = \text{attach-rop } r \text{ } xb$ **in** exI , *auto simp:Seq-def*)

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

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

3.1.2 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
         $x \in L \{Trn Y (CHAR c) \mid Y c. Y \in UNIV // (\approx Lang) \wedge Y ;; \{[c]\} \subseteq X\}$ 
        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  $L xrhs \subseteq X$  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  $\{Trn Y (CHAR c) \mid Y c. Y \in CS \wedge Y ;; \{[c]\} \subseteq X\}$  (is finite ?A)
  proof –
    def  $S \equiv \{(Y, c) \mid Y c. Y \in CS \wedge Y ;; \{[c]\} \subseteq X\}$ 
    def  $h \equiv \lambda (Y, c). Trn Y (CHAR c)$ 
    have finite  $(CS \times (UNIV::char set))$  using finite by auto
    hence finite S using S-def
    by (rule-tac B = CS × UNIV in finite-subset, auto)
    moreover have  $?A = h \text{ ' } S$  by (auto simp: S-def h-def image-def)
    ultimately show ?thesis
  qed

```

```

    by auto
  qed
  thus ?thesis by (simp add:init-rhs-def)
qed

```

```

lemma init-ES-satisfy-Inv:
  assumes finite-CS: finite (UNIV // ( $\approx$ Lang))
  shows Inv (eqs (UNIV // ( $\approx$ Lang)))
proof –
  have finite (eqs (UNIV // ( $\approx$ Lang))) using finite-CS
    by (simp add:eqs-def)
  moreover have distinct-equas (eqs (UNIV // ( $\approx$ Lang)))
    by (simp add:distinct-equas-def eqs-def)
  moreover have ardenable (eqs (UNIV // ( $\approx$ Lang)))
    by (auto simp add:ardenable-def eqs-def init-rhs-def rhs-nonempty-def del:L-rhs.simps)
  moreover have valid-egns (eqs (UNIV // ( $\approx$ Lang)))
    using l-eq-r-in-egs by (simp add:valid-egns-def)
  moreover have non-empty (eqs (UNIV // ( $\approx$ Lang)))
    by (auto simp:non-empty-def eqs-def quotient-def Image-def str-eq-rel-def)
  moreover have finite-rhs (eqs (UNIV // ( $\approx$ Lang)))
    using finite-init-rhs[OF finite-CS]
    by (auto simp:finite-rhs-def eqs-def)
  moreover have self-contained (eqs (UNIV // ( $\approx$ 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

```

3.1.3 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 $L(\text{arden-variate } X \text{ rhs}) = (B ;; A\star)$ (**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*:
 $\text{finite } rhs \implies \text{finite } (\text{append-rhs-rexp } rhs \ r)$
by (*auto simp:append-rhs-rexp-def*)

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

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

lemma *nonempty-set-sub*:
 $rhs\text{-nonempty } rhs \implies rhs\text{-nonempty } (rhs - A)$
by (*auto simp:rhs-nonempty-def*)

lemma *nonempty-set-union*:
 $\llbracket rhs\text{-nonempty } rhs; rhs\text{-nonempty } rhs \rrbracket \implies rhs\text{-nonempty } (rhs \cup rhs')$
by (*auto simp:rhs-nonempty-def*)

lemma *arden-variate-keeps-nonempty*:
 $rhs\text{-nonempty } rhs \implies rhs\text{-nonempty } (\text{arden-variate } X \text{ rhs})$
by (*simp only:arden-variate-def append-keeps-nonempty nonempty-set-sub*)

lemma *rhs-subst-keeps-nonempty*:
 $\llbracket rhs\text{-nonempty } rhs; rhs\text{-nonempty } xrhs \rrbracket \implies rhs\text{-nonempty } (rhs\text{-subst } rhs \ X \ xrhs)$
by (*simp only:rhs-subst-def append-keeps-nonempty nonempty-set-union nonempty-set-sub*)

lemma *rhs-subst-keeps-eq*:
assumes *substor*: $X = L \ xrhs$
and *finite*: $\text{finite } rhs$
shows $L(\text{rhs-subst } rhs \ X \ xrhs) = L \ rhs$ (**is** $?Left = ?Right$)
proof –
def $A \equiv L(\text{rhs} - \text{items-of } rhs \ X)$
have $?Left = A \cup L(\text{append-rhs-rexp } xrhs \ (\text{rexp-of } rhs \ X))$
by (*simp only:rhs-subst-def L-rhs-union-distrib A-def*)
moreover have $?Right = A \cup L(\text{items-of } rhs \ X)$
proof –

have $rhs = (rhs - items\text{-}of\ rhs\ X) \cup (items\text{-}of\ rhs\ X)$ **by** (*auto simp:items-of-def*)
thus $?thesis$ **by** (*simp only:L-rhs-union-distrib A-def*)
qed
moreover **have** $L (append\text{-}rhs\text{-}rexp\ xrhs (rexp\text{-}of\ rhs\ X)) = L (items\text{-}of\ rhs\ X)$
using *finite subst* **by** (*simp only:lang-of-append-rhs lang-of-rexp-of*)
ultimately **show** $?thesis$ **by** *simp*
qed

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

lemma *eqs-subst-keeps-finite*:
assumes *finite:finite (ES:: (string set \times rhs-item set) set)*
shows *finite (eqs-subst ES Y yrhs)*
proof –
have $finite\ \{(Ya, rhs\text{-}subst\ yrhsa\ Y\ yrhs) \mid Ya\ yrhsa. (Ya, yrhsa) \in ES\}$
(is finite ?A)

proof–
def $eqns' \equiv \{(Ya::string\ set), yrhsa \mid Ya\ yrhsa. (Ya, yrhsa) \in ES\}$
def $h \equiv \lambda ((Ya::string\ set), yrhsa). (Ya, rhs\text{-}subst\ yrhsa\ Y\ yrhs)$
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*:
 $\llbracket finite\text{-}rhs\ ES; finite\ yrhs \rrbracket \implies finite\text{-}rhs\ (eqs\text{-}subst\ ES\ Y\ yrhs)$
by (*auto intro:rhs-subst-keeps-finite-rhs simp add:eqs-subst-def finite-rhs-def*)

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

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

lemma *rhs-subst-updates-cls*:
 $X \notin classes\text{-}of\ xrhs \implies$

$classes-of (rhs-subst rhs X xrhs) = classes-of rhs \cup classes-of xrhs - \{X\}$
apply (*simp only:rhs-subst-def append-rhs-keeps-cl*
 $classes-of-union-distrib[THEN sym]$)
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 $self-contained (eqs-subst ES Y (arden-variate Y yrhs))$
(**is self-contained** ? B)

proof –

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

proof –

have $classes-of xrhs' \subseteq$
 $classes-of xrhs \cup classes-of (arden-variate Y yrhs) - \{Y\}$

proof –

have $Y \notin classes-of (arden-variate Y yrhs)$
using *arden-variate-removes-cl* **by** *simp*
thus ?*thesis* **using** $xrhs-xrhs'$ **by** (*auto simp:rhs-subst-updates-cl*)
qed

moreover have $classes-of xrhs \subseteq lefts-of ES \cup \{Y\}$ **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 $classes-of (arden-variate Y yrhs) \subseteq lefts-of ES \cup \{Y\}$
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 $Inv (eqs-subst ES Y (arden-variate Y yrhs))$

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-eqns-def, blast)
have distinct-equas (eqs-subst ES Y (arden-variate Y yrhs))
  using Inv-ES
  by (auto simp:distinct-equas-def eqs-subst-def Inv-def)
moreover have finite (eqs-subst ES Y (arden-variate Y yrhs))
  using Inv-ES by (simp add:Inv-def eqs-subst-keeps-finite)
moreover have finite-rhs (eqs-subst ES Y (arden-variate Y yrhs))
proof-
  have finite-rhs ES using Inv-ES
    by (simp add:Inv-def finite-rhs-def)
  moreover have finite (arden-variate Y 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 (eqs-subst ES Y (arden-variate Y 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* ($ES::(\text{string set} \times \text{rhs-item set}) \text{ set}$)

shows $\text{card} (\text{eqs-subst } ES \ Y \ \text{yrhs}) \leq \text{card } ES$

proof –

def $f \equiv \lambda x. ((fst \ x)::\text{string set}, \text{rhs-subst } (snd \ x) \ Y \ \text{yrhs})$

have $\text{eqs-subst } ES \ Y \ \text{yrhs} = f \ ' \ ES$

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*:

$(X, \text{xrhs}) \in ES \implies \exists \text{xrhs}'. (X, \text{xrhs}') \in (\text{eqs-subst } ES \ Y \ \text{yrhs})$

by (*auto simp: eqs-subst-def*)

lemma *card-noteq-1-has-more*:

assumes *card*: $\text{card } S \neq 1$

and *e-in*: $e \in S$

and *finite*: *finite* S

obtains e' where $e' \in S \wedge e \neq e'$

proof –

have $\text{card} (S - \{e\}) > 0$

proof –

have $\text{card } S > 1$ using *card e-in finite*

by (*case-tac card S, auto*)

thus *?thesis* using *finite e-in* by *auto*

qed

hence $S - \{e\} \neq \{\}$ using *finite* by (*rule-tac notI, simp*)

thus $(\bigwedge e'. e' \in S \wedge e \neq e' \implies \text{thesis}) \implies \text{thesis}$ by *auto*

qed

lemma *iteration-step*:

assumes *Inv-ES*: *Inv* ES

and *X-in-ES*: $(X, \text{xrhs}) \in ES$

and *not-T*: $\text{card } ES \neq 1$

shows $\exists ES'. (\text{Inv } ES' \wedge (\exists \text{xrhs}'. (X, \text{xrhs}') \in ES')) \wedge$
 $(\text{card } ES', \text{card } ES) \in \text{less-than} (\text{is } \exists ES'. ?P \ ES')$

proof –

have *finite-ES*: *finite* ES using *Inv-ES* by (*simp add: Inv-def*)

then obtain $Y \ \text{yrhs}$

where *Y-in-ES*: $(Y, \text{yrhs}) \in ES$ and *not-eq*: $(X, \text{xrhs}) \neq (Y, \text{yrhs})$

using *not-T X-in-ES* by (*drule-tac card-noteq-1-has-more, auto*)

def $ES' == ES - \{(Y, \text{yrhs})\}$

let *?ES''* = *eqs-subst* $ES' \ Y$ (*arden-variate* $Y \ \text{yrhs}$)

have *?P ?ES''*

proof –

have *Inv ?ES''* using *Y-in-ES Inv-ES*

by (rule-tac eqs-subst-satisfy-Inv, simp add:ES'-def insert-absorb)
 moreover have $\exists xrhs'. (X, xrhs') \in ?ES''$ using not-eq X-in-ES
 by (rule-tac ES = ES' in eqs-subst-cls-remains, auto simp add:ES'-def)
 moreover have (card ?ES'', card ES) \in less-than
 proof –
 have finite ES' using finite-ES ES'-def by auto
 moreover have card ES' < card ES using finite-ES Y-in-ES
 by (auto simp:ES'-def card-gt-0-iff intro:diff-Suc-less)
 ultimately show ?thesis
 by (auto dest:eqs-subst-card-le elim:le-less-trans)
 qed
 ultimately show ?thesis by simp
 qed
 thus ?thesis by blast
 qed

3.1.4 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 history: Inv ES
 and X-in-ES: $\exists xrhs. (X, xrhs) \in ES$
 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 ... = 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 ... = 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 []  $\notin$  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-eqcl-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, 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, 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, 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::rexpr). Lang = L reg
proof -
  have  $\forall X \in$  (UNIV // ( $\approx$ Lang)).  $\exists$  (reg::rexpr). X = L reg
    using finite-CS every-reqcl-has-reg by blast
  then obtain f
    where f-prop:  $\forall X \in$  (UNIV // ( $\approx$ Lang)). X = L ((f X)::rexpr)
    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 ... = 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

```

4 Direction: regular language \Rightarrow finite partition

4.1 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:  $\bigwedge m n. \text{tag } m = \text{tag } (n::\text{string}) \implies m \approx \text{lang } n$ 
  and range: finite (range tag)
  shows finite (UNIV // ( $\approx \text{lang}$ ))
proof (rule-tac f = (op ‘) tag in finite-imageD)
  show finite (op ‘ tag ‘ UNIV //  $\approx \text{lang}$ ) using range
  apply (rule-tac B = Pow (tag ‘ UNIV) in finite-subset)
  by (auto simp add:image-def Pow-def)
next
  show inj-on (op ‘ tag) (UNIV //  $\approx \text{lang}$ )
  proof–
  { fix X Y
    assume X-in:  $X \in \text{UNIV} // \approx \text{lang}$ 
    and Y-in:  $Y \in \text{UNIV} // \approx \text{lang}$ 
    and tag-eq: tag ‘ X = tag ‘ 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

```

4.2 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 // ( $\approx \{\emptyset\}$ )  $\subseteq \{\{\emptyset\}, \text{UNIV} - \{\emptyset\}\}$ 
proof
  fix x
  assume  $x \in \text{UNIV} // \approx \{\emptyset\}$ 
  then obtain y where  $h: x = \{z. (y, z) \in \approx \{\emptyset\}\}$ 
  unfolding quotient-def Image-def by blast

```

```

show  $x \in \{\{\}\}, UNIV - \{\{\}\}$ 
proof (cases  $y = \{\}$ )
  case True with  $h$ 
    have  $x = \{\{\}\}$  by (auto simp:str-eq-rel-def)
    thus ?thesis by simp
  next
    case False with  $h$ 
    have  $x = UNIV - \{\{\}\}$  by (auto simp:str-eq-rel-def)
    thus ?thesis by simp
qed
qed

```

lemma quot-char-subset:

```

UNIV // ( $\approx\{\{c\}\}$ )  $\subseteq \{\{\{\}\},\{c\}, UNIV - \{\{\}, [c]\}\}$ 
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  $x \in \{\{\{\}\},\{c\}, UNIV - \{\{\}, [c]\}\}$ 
  proof -
    { assume  $y = \{\}$  hence  $x = \{\{\}\}$  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

```

4.3 The case for SEQ

definition

```

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

```

lemma tag-str-seq-range-finite:

```

[[finite (UNIV //  $\approx L_1$ ); finite (UNIV //  $\approx L_2$ )]
   $\implies$  finite (range (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-in-seq*: $x @ z \in L_1 ;; L_2$

and *tag-xy*: *tag-str-SEQ* $L_1 L_2 x = \text{tag-str-SEQ } L_1 L_2 y$

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-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 $\{\approx_{L_2} \text{ “ } \{x - xa\} \mid xa. xa \leq x \wedge xa \in L_1 \} =$

$\{\approx_{L_2} \text{ “ } \{y - xa\} \mid xa. xa \leq y \wedge xa \in L_1 \}$

(**is** *?Left = ?Right*)

using $h1$ *tag-xy* **by** (*auto simp:tag-str-SEQ-def*)

moreover **have** $\approx_{L_2} \text{ “ } \{x - xa\} \in \text{?Left}$ **using** $h1 h2$ **by** *auto*

ultimately **have** $\approx_{L_2} \text{ “ } \{x - xa\} \in \text{?Right}$ **by** *simp*

thus *?thesis* **by** (*auto simp:Image-def str-eq-rel-def str-eq-def*)

qed

with *prems* **show** *?thesis* **by** (*auto simp:str-eq-rel-def str-eq-def*)

qed

hence $y @ z \in L_1 ;; L_2$ **by** (*erule-tac prefixE, auto simp:Seq-def*)

```

} moreover {
  fix za
  assume h1: za ≤ z and h2: (x @ za) ∈ L1 and h3: z - za ∈ L2
  hence y @ za ∈ L1
  proof-
    have ≈L1 “ {x} = ≈L1 “ {y}
      using h1 tag-xy by (auto simp:tag-str-SEQ-def)
    with h2 show ?thesis
      by (auto simp:Image-def str-eq-rel-def str-eq-def)
  qed
  with h1 h3 have y @ z ∈ L1 ;; L2
    by (drule-tac A = L1 in seq-intro, auto elim:prefixE)
}
ultimately show ?thesis by blast
qed
} thus tag-str-SEQ L1 L2 m = tag-str-SEQ L1 L2 n ⇒ m ≈(L1 ;; L2) n
  by (auto simp add: str-eq-def str-eq-rel-def)
qed

```

lemma *quot-seq-finiteI*:
 $\llbracket \text{finite } (UNIV // \approx L_1); \text{finite } (UNIV // \approx L_2) \rrbracket$
 $\implies \text{finite } (UNIV // \approx (L_1 ;; L_2))$
apply (rule-tac tag = tag-str-SEQ L₁ L₂ in tag-finite-imageD)
by (auto intro:tag-str-SEQ-injI elim:tag-str-seq-range-finite)

4.4 The case for ALT

definition

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

lemma *quot-union-finiteI*:

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

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

shows $\text{finite } (UNIV // \approx (L_1 \cup L_2))$

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

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

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

next

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

4.5 The case for STAR

This turned out to be the trickiest case.

definition

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

```

lemma finite-set-has-max:  $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies$ 
   $(\exists \text{ max} \in A. \forall a \in A. f a \leq (f \text{ max} :: \text{nat}))$ 
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:  $\text{max} \in A$ 
    and h2:  $\forall a \in A. f a \leq f \text{ max}$  by blast
  show ?thesis
  proof (cases f a ≤ f max)
    assume  $f a \leq f \text{ max}$ 
    with h1 h2 show ?thesis by (rule-tac x = max in beXI, auto)
  next
    assume  $\neg (f a \leq f \text{ max})$ 
    thus ?thesis using h2 by (rule-tac x = a in beXI, auto)
  qed
qed
qed

```

```

lemma finite-strict-prefix-set:  $\text{finite } \{xa. xa < (x::\text{string})\}$ 
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:
   $\text{finite } (\text{UNIV} // \approx_{L_1}) \implies \text{finite } (\text{range } (\text{tag-str-STAR } L_1))$ 
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:
   $\text{tag-str-STAR } L_1 m = \text{tag-str-STAR } L_1 n \implies m \approx_{(L_1^*)} n$ 
proof -
  { fix x y z
    assume xz-in-star:  $x @ z \in L_1^*$ 
    and tag-xy:  $\text{tag-str-STAR } L_1 x = \text{tag-str-STAR } L_1 y$ 
    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:  $x\text{-max} < x$ 
  and h2:  $x\text{-max} \in L_1\star$ 
  and h3:  $(x - x\text{-max}) @ z \in L_1\star$ 
  and h4:  $\forall xa < x. xa \in L_1\star \wedge (x - xa) @ z \in L_1\star$ 
     $\longrightarrow \text{length } xa \leq \text{length } x\text{-max}$ 
proof-
let ?S =  $\{xa. xa < x \wedge xa \in L_1\star \wedge (x - xa) @ z \in L_1\star\}$ 
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  $\exists \text{max} \in ?S. \forall a \in ?S. \text{length } a \leq \text{length } \text{max}$ 
  using finite-set-has-max by blast
with prems show ?thesis by blast
qed
obtain ya
  where h5:  $ya < y$  and h6:  $ya \in L_1\star$  and h7:  $(x - x\text{-max}) \approx_{L_1} (y - ya)$ 
proof-
from tag-xy have  $\{\approx_{L_1} \text{“ } \{x - xa\} | xa. xa < x \wedge xa \in L_1\star \} =$ 
   $\{\approx_{L_1} \text{“ } \{y - xa\} | xa. xa < y \wedge xa \in L_1\star \}$  (is ?left = ?right)
  by (auto simp: tag-str-STAR-def)
moreover have  $\approx_{L_1} \text{“ } \{x - x\text{-max}\} \in ?left$  using h1 h2 by auto
ultimately have  $\approx_{L_1} \text{“ } \{x - x\text{-max}\} \in ?right$  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-max' =  $x\text{-max} @ a$ 
  have ?x-max' < x
  using np h1 by (clarsimp simp: strict-prefix-def diff-prefix)
  moreover have ?x-max'  $\in L_1\star$ 

```

```

    using a-in h2 by (simp add:star-intro3)
    moreover have (x - ?x-max') @ z ∈ L1★
    using b-egs b-in np h1 by (simp add:diff-diff-appd)
    moreover have ¬ (length ?x-max' ≤ length x-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-max) @ za ∈ L1
  using a-in by (auto elim:prefixE)
from x-za h7 have (y - ya) @ za ∈ L1
  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 L1 m = tag-str-STAR L1 n ⇒ m ≈(L1★) n
  by (auto simp add:str-eq-def str-eq-rel-def)
qed

```

lemma *quot-star-finiteI*:

```

finite (UNIV // ≈L1) ⇒ finite (UNIV // ≈(L1★))
apply (rule-tac tag = tag-str-STAR L1 in tag-finite-imageD)
by (auto intro:tag-str-STAR-injI elim:tag-str-star-range-finite)

```

4.6 The main lemma

lemma *easier-direction*:

```

Lang = L (r::rexp) ⇒ finite (UNIV // (≈Lang))
proof (induct arbitrary:Lang rule:rexp.induct)
  case NULL
  have UNIV // (≈{}) ⊆ {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 // (≈{[]}) ⊆ {{[]}, UNIV - {[]}}
    by (rule quot-empty-subset)
  with prems show ?case by (auto intro:finite-subset)
next
  case (CHAR c)
  have UNIV // (≈{[c]}) ⊆ {{[]},{[c]}, UNIV - {[], [c]}}
    by (rule quot-char-subset)
  with prems show ?case by (auto intro:finite-subset)
next
  case (SEQ r1 r2)

```

```

have  $\llbracket \text{finite } (UNIV // \approx(L r_1)); \text{finite } (UNIV // \approx(L r_2)) \rrbracket$ 
       $\implies \text{finite } (UNIV // \approx(L r_1 ;; L r_2))$ 
  by (erule quot-seq-finiteI, simp)
with prems show ?case by simp
next
case (ALT r1 r2)
have  $\llbracket \text{finite } (UNIV // \approx(L r_1)); \text{finite } (UNIV // \approx(L r_2)) \rrbracket$ 
       $\implies \text{finite } (UNIV // \approx(L r_1 \cup L r_2))$ 
  by (erule quot-union-finiteI, simp)
with prems show ?case by simp
next
case (STAR r)
have  $\text{finite } (UNIV // \approx(L r))$ 
       $\implies \text{finite } (UNIV // \approx((L r)\star))$ 
  by (erule quot-star-finiteI)
with prems show ?case by simp
qed

end

```