tphols-2011

By xingyuan

January 27, 2011

Contents

theory *Myhill*
  imports *Main List-Prefix Prefix-subtract Prelude*
begin

1 Preliminary definitions

types *lang* = *string set*

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

definition *Seq* :: *lang* ⇒ *lang* ⇒ *lang* (- ;; - [*100*,*100*] *100*)
where
  L1 ;; *L2* = {*s1* @ *s2* | *s1 s2*. *s1* ∈ *L1* ∧ *s2* ∈ *L2*}

Transitive closure of language *L*.

inductive-set
  Star :: *string set* $\Rightarrow$ *string set* (-$\star$ [*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⋆
  have A⋆ = {[]} ∪ A⋆ ;; A
    by (auto simp:Seq-def star-intro3 star-decom')
  then have B ;; A⋆ = B ;; ({[]} ∪ A⋆ ;; A)
    unfolding Seq-def by simp
  also have ... = B ∪ B ;; (A⋆ ;; A)
    unfolding Seq-def by auto
  also have ... = B ∪ (B ;; A⋆) ;; A
    by (simp only:seq-assoc)
  finally show X = X ;; A ∪ B
    using eq by blast
next
  assume eq': X = X ;; A ∪ B
  hence c1': ⋀ x. x ∈ B ⟹ x ∈ X
    and c2': ⋀ x y. ⟦x ∈ X; y ∈ A⟧ ⟹ x @ y ∈ X
    using Seq-def by auto
  show X = B ;; A⋆
  proof
    show B ;; A⋆ ⊆ X
    proof−
      { fix x y
        have ⟦y ∈ A⋆; x ∈ X⟧ ⟹ x @ y ∈ 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 ⊆ B ;; A⋆
    proof−
      { fix x
        have x ∈ X ⟹ x ∈ B ;; A⋆
        proof (induct x taking:length rule:measure-induct)
          fix z
          assume hyps:
            ∀ y. length y < length z ⟶ y ∈ X ⟶ y ∈ B ;; A⋆
            and z-in: z ∈ X
          show z ∈ B ;; A⋆
          proof (cases z ∈ B)
            case True thus ?thesis by (auto simp:Seq-def start)
          next
            case False hence z ∈ X ;; A using eq' z-in by auto
            then obtain za zb where za-in: za ∈ X
              and zab: z = za @ zb ∧ zb ∈ A and zbne: zb ≠ []
              using nemp unfolding Seq-def by blast
            from zbne zab have length za < length z by auto
            with za-in hyps have za ∈ B ;; A⋆ by blast
            hence za @ zb ∈ 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 ⇒ string set
```

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

```
overloading L-rexp ≡ L::  rexp ⇒ string set
begin
fun
  L-rexp :: rexp ⇒ 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) ∪ (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 ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b
where
  folds f z S ≡ SOME x. 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]:
```

finite rs $\Longrightarrow$ *L* (*folds ALT NULL rs*) = $\bigcup$ (*L* ' *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*]:
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 ($\approx$- [*100*] *100*)
where
$\approx Lang \equiv \{(x, y).\ (\forall z.\ x\ @\ z \in Lang \longleftrightarrow y\ @\ z \in Lang)\}$

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

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

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

lemma *lang-is-union-of-finals*:
Lang = $\bigcup$ *finals*(*Lang*)
proof
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
show $\bigcup$ (*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* ⇒ *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 λ). Similary, the second equation tells how the strings inside X_1 are composed.

$$\begin{aligned} X_0 &= X_0 b + X_1 a + \lambda \\ X_1 &= X_0 a + X_1 b \end{aligned} \tag{1}$$

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 (1), the other kind corresponds to transitions from one one equivalent class to another, like the $X_0 b, X_1 a$ etc.

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

In this formalization, pure regular expressions like λ is repsented by $Lam(EMPTY)$, while transitions like $X_0 a$ 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* ⇒ *rexp*
where *the-r* (*Lam r*) = *r*

fun *the-Trn*:: *rhs-item* ⇒ (*string set* × *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* ≡ *L*:: *rhs-item* ⇒ *string set*
begin
  fun *L-rhs-e*:: *rhs-item* ⇒ *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* ≡ *L*:: *rhs-item set* ⇒ *string set*
begin
  fun *L-rhs*:: *rhs-item set* ⇒ *string set*
  where *L-rhs rhs* = ⋃ (*L* ' *rhs*)
end

Given a set of equivalent classses *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 ≡

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

In the definition of *init-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 λ in (1).

With the help of *init-rhs*, the equitional system descrbing 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-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-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
$rexp\text{-}of\ rhs\ X \equiv folds\ ALT\ NULL\ ((snd\ o\ the\text{-}Trn)\ `\ items\text{-}of\ rhs\ X)$

The following *lam-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-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 compute the regular expression corresponds to *rhs*.

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

The following *attach-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
$attach\text{-}rexp\ rexp'\ (Lam\ rexp) \quad = Lam\ (SEQ\ rexp\ rexp')$
$\mid attach\text{-}rexp\ rexp'\ (Trn\ X\ rexp) = Trn\ X\ (SEQ\ rexp\ rexp')$

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

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

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
arden-variate X rhs ≡
append-rhs-rexp (*rhs* − *items-of rhs X*) (*STAR* (*rexp-of rhs X*))

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

definition
rhs-subst rhs X xrhs ≡
(*rhs* − (*items-of rhs X*)) ∪ (*append-rhs-rexp xrhs* (*rexp-of rhs X*))

Suppose the equation defining X is $X = xrhs$, the follwing *eqs-subst ES X xrhs* substitute *xrhs* into every equation of the equational system *ES*.

definition
eqs-subst ES X xrhs ≡ {(*Y*, *rhs-subst yrhs X xrhs*) | *Y yrhs*. (*Y*, *yrhs*) ∈ *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*. ⟦*P e*; ¬ *Q e*⟧ ⟹ (∃ *e'*. *P e'* ∧ (*f*(*e'*), *f*(*e*)) ∈ *less-than*)
shows *pe*: *P e* ⟶ (∃ *e'*. *P e'* ∧ *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*]:
∀ *y*. (*y*, *x*) ∈ *inv-image less-than f* ⟶ *P y* ⟶ (∃ *e'*. *P e'* ∧ *Q e'*)
and *px*: *P x*
show ∃ *e'*. *P e'* ∧ *Q e'*
proof(*cases Q x*)
assume *Q x* **with** *px* **show** *?thesis* **by** *blast*
next
assume *nq*: ¬ *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 invaiant 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 onece in ES.

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

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

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

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

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

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

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

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

The following *finite-rhs ES* requires every equation in *rhs* be finite.

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

The following *classes-of rhs* returns all variables (or equivalent classes) occuring in *rhs*.

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

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

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

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

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

The invariant *Inv*(*ES*) is a conjunction of all the previously defined constaints.

definition
Inv ES $\equiv$ *valid-eqns ES* $\wedge$ *finite ES* $\wedge$ *distinct-equas ES* $\wedge$ *ardenable ES* $\wedge$
non-empty ES $\wedge$ *finite-rhs ES* $\wedge$ *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 $(A::\textit{rhs-item set}) \cup L\ B = L\ (A \cup B)$
by *simp*

lemma *finite-snd-Trn*:
assumes *finite*:*finite rhs*
shows *finite* $\{r_2.\ \textit{Trn}\ Y\ r_2 \in \textit{rhs}\}$ (**is** *finite ?B*)
proof−
def $\textit{rhs}' \equiv \{e \in \textit{rhs}.\ \exists\ r.\ e = \textit{Trn}\ Y\ r\}$
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 $[] \notin L$ (*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*!]:
$P\ (\textit{Trn}\ X\ r) \Longrightarrow (\exists\, a.\ (\exists\, r.\ a = \textit{Trn}\ X\ r \wedge P\ a))$ **by** *auto*

lemma *finite-items-of*:
finite rhs $\Longrightarrow$ *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* $((snd \circ the\text{-}Trn)$ ‘ *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*:
$[\![s \in X;\ X \in UNIV\ //\ (\approx Lang)]\!] \Longrightarrow X = (\approx Lang)$ ‘‘ $\{s\}$
by (*auto simp*:*quotient-def Image-def str-eq-rel-def*)

lemma *every-eqclass-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 $Y \equiv (\approx Lang)$ `` $\{s\}$
  have $Y \in UNIV$ // $(\approx Lang)$
    unfolding *Y-def quotient-def* **by** *auto*
  moreover
  have $X = (\approx Lang)$ `` $\{s @ [c]\}$
    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-eqclass-has-transition* **by** *blast*
      hence
        $x \in L\ \{Trn\ Y\ (CHAR\ c)|\ Y\ c.\ Y \in UNIV\ //\ (\approx Lang) \wedge Y\ ;;\ \{[c]\} \subseteq X\}$
        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*) $|$*Y c*. *Y* $\in$ *CS* $\wedge$ *Y* ;; $\{[c]\} \subseteq X\}$ (**is** *finite ?A*)
proof −
def $S \equiv \{(Y,\ c)|\ Y\ c.\ Y \in CS \wedge Y$;; $\{[c]\} \subseteq X\}$
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* $\times$ *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* // ($\approx$*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-eqns* (*eqs* (*UNIV* // ($\approx$*Lang*)))
using *l-eq-r-in-eqs* **by** (*simp add*:*valid-eqns-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

2.1.3 Interation 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*: $[] \notin L$ (*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* ∪ *b* **by** (*auto simp*:*b-def items-of-def*)
hence *L rhs* = *L*(*items-of rhs X* ∪ *b*) **by** *simp*
hence *L rhs* = *L*(*items-of rhs X*) ∪ *B* **by** (*simp only*:*L-rhs-union-distrib B-def*)
with *lang-of-rexp-of*
have *L rhs* = *X* ;; *A* ∪ *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 ⟹ *finite* (*append-rhs-rexp rhs r*)
by (*auto simp*:*append-rhs-rexp-def*)

lemma *arden-variate-keeps-finite*:
finite rhs ⟹ *finite* (*arden-variate X rhs*)
by (*auto simp*:*arden-variate-def append-keeps-finite*)

lemma *append-keeps-nonempty*:
rhs-nonempty rhs ⟹ *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 ⟹ *rhs-nonempty* (*rhs* − *A*)
by (*auto simp*:*rhs-nonempty-def*)

lemma *nonempty-set-union*:
⟦*rhs-nonempty rhs*; *rhs-nonempty rhs'*⟧ ⟹ *rhs-nonempty* (*rhs* ∪ *rhs'*)
by (*auto simp*:*rhs-nonempty-def*)

lemma *arden-variate-keeps-nonempty*:
rhs-nonempty rhs ⟹ *rhs-nonempty* (*arden-variate X rhs*)

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

lemma *rhs-subst-keeps-nonempty*:
  $[\![$*rhs-nonempty rhs*; *rhs-nonempty xrhs*$]\!] \Longrightarrow$ *rhs-nonempty* (*rhs-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*: *finite rhs*
  shows *L* (*rhs-subst rhs X xrhs*) = *L rhs* (**is** *?Left* = *?Right*)
proof−
  def $A \equiv L$ (*rhs* − *items-of rhs X*)
  have *?Left* = $A \cup L$ (*append-rhs-rexp xrhs* (*rexp-of rhs X*))
    by (*simp only*:*rhs-subst-def L-rhs-union-distrib A-def*)
  moreover have *?Right* = $A \cup L$ (*items-of rhs X*)
  proof−
   have *rhs* = (*rhs* − *items-of rhs X*) ∪ (*items-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-rhs-rexp xrhs* (*rexp-of rhs X*)) = *L* (*items-of 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*:
  $[\![$*finite rhs*; *finite yrhs*$]\!] \Longrightarrow$ *finite* (*rhs-subst rhs Y yrhs*)
by (*auto simp*:*rhs-subst-def append-keeps-finite*)

lemma *eqs-subst-keeps-finite*:
  assumes *finite*:*finite* (*ES*:: (*string set* × *rhs-item set*) *set*)
  shows *finite* (*eqs-subst ES Y yrhs*)
proof −
  have *finite* {(*Ya*, *rhs-subst yrhsa Y yrhs*) |*Ya yrhsa*. (*Ya*, *yrhsa*) ∈ *ES*}
                                                            (**is** *finite ?A*)
  proof−
    def *eqns′* ≡ {((*Ya*::*string set*), *yrhsa*)| *Ya yrhsa*. (*Ya*, *yrhsa*) ∈ *ES*}
    def $h \equiv \lambda$ ((*Ya*::*string set*), *yrhsa*). (*Ya*, *rhs-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*:
  $[\![$*finite-rhs ES*; *finite yrhs*$]\!] \Longrightarrow$ *finite-rhs* (*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-cls*:
  classes-of (*append-rhs-rexp rhs r*) = *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*:
  classes-of (*arden-variate Y yrhs*) = *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-cls*:
  lefts-of (*eqs-subst ES Y yrhs*) = *lefts-of ES*
by (*auto simp*:*lefts-of-def eqs-subst-def*)

lemma *rhs-subst-updates-cls*:
  $X \notin$ *classes-of xrhs* $\Longrightarrow$
      classes-of (*rhs-subst rhs X xrhs*) = *classes-of rhs* ∪ *classes-of xrhs* − {*X*}
apply (*simp only*:*rhs-subst-def append-rhs-keeps-cls*
                              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* ∪ {(*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′*) ∈ *?B*
    then obtain *xrhs*
      where *xrhs-xrhs′*: *xrhs′* = *rhs-subst xrhs Y* (*arden-variate Y yrhs*)
      and *X-in*: (*X*, *xrhs*) ∈ *ES* **by** (*simp add*:*eqs-subst-def*, *blast*)
    have *classes-of xrhs′* ⊆ *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′* ⊆ *lefts-of ES*
      proof−
        have *classes-of xrhs′* ⊆
                    classes-of xrhs ∪ *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-cls*)
        qed
        moreover have *classes-of xrhs* ⊆ *lefts-of ES* ∪ {*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*) ⊆ *lefts-of ES* ∪ {*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 ∪ {(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) ∈ 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 $f \equiv \lambda\ x.$ ((*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*) ∈ *ES* $\Longrightarrow \exists$ *xrhs*$'$. (X, *xrhs*$'$) ∈ (*eqs-subst ES Y yrhs*)
by (*auto simp*:*eqs-subst-def*)

lemma *card-noteq-1-has-more*:
  assumes *card*:*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 *card* $(S - \{e\}) > 0$
  proof −
    have *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 $(\bigwedge e'.\ e' \in S \wedge e \neq e' \Longrightarrow thesis) \Longrightarrow thesis$ **by** *auto*
qed

```
lemma iteration-step:
  assumes Inv-ES: Inv ES
  and     X-in-ES: (X, xrhs) ∈ ES
  and     not-T: card ES ≠ 1
  shows ∃ ES'. (Inv ES' ∧ (∃ xrhs'.(X, xrhs') ∈ ES')) ∧
              (card ES', card ES) ∈ less-than (is ∃ ES'. ?P ES')
proof −
  have finite-ES: finite ES using Inv-ES by (simp add:Inv-def)
  then obtain Y yrhs
    where Y-in-ES: (Y, yrhs) ∈ ES and not-eq: (X, xrhs) ≠ (Y, yrhs)
    using not-T X-in-ES by (drule-tac card-noteq-1-has-more, auto)
  def ES' == ES − {(Y, yrhs)}
  let ?ES'' = eqs-subst ES' Y (arden-variate Y 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 ∃xrhs'. (X, xrhs') ∈ ?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) ∈ 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
```

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 history: Inv ES
  and     X-in-ES: ∃ xrhs. (X, xrhs) ∈ ES
  shows
  ∃ ES'. (Inv ES' ∧ (∃ xrhs'. (X, xrhs') ∈ ES')) ∧ card ES' = 1
                                                    (is ∃ 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 ∃ (r::rexp). L r = X (is ∃ 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 [] ∉ 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 // (≈Lang))
  and X-in-CS: X ∈ (UNIV // (≈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*::*rexp*). *Lang* = *L reg*
proof −
have $\forall$ *X* $\in$ (*UNIV* // ($\approx$*Lang*)). $\exists$ (*reg*::*rexp*). *X* = *L reg*
using *finite-CS every-eqcl-has-reg* **by** *blast*
then obtain *f*
where *f-prop*: $\forall$ *X* $\in$ (*UNIV* // ($\approx$*Lang*)). *X* = *L* ((*f X*)::*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 ... = *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 partitions*

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 (- ≈- -)
where
  $x \approx Lang\ y \equiv (x,\ y) \in (\approx Lang)$

The very basic scheme to show the finiteness of the partion generated by a language *Lang* is by attaching tags to every string. The set of tags are carefully choosen to make it finite. If it can be proved that strings with the same tag are equivlent 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 equivalence classes (i.e. ((*op* ‘) *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*.

Theorems *tag-image-injI* **and** *finite-tag-imageI* **do not exist. Can this comment be deleted?** **COMMENT**

lemma *tag-finite-imageD*:
  fixes *L1*::*lang*
  assumes *str-inj*: $\bigwedge$ *m n. tag m = tag n* $\Longrightarrow$ *m* ≈*L1 n*
  and *range*: *finite* (*range tag*)
  shows *finite* (*UNIV* // ≈*L1*)
proof (*rule-tac f* = (*op* ‘) *tag* **in** *finite-imageD*)
  show *finite* (*op* ‘ *tag* ‘ *UNIV* // ≈*L1*) **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* // ≈*L1*)
  proof−
    { fix *X Y*
      assume *X-in*: *X* ∈ *UNIV* // ≈*L1*
        and *Y-in*: *Y* ∈ *UNIV* // ≈*L1*
        and *tag-eq*: *tag* ‘ *X* = *tag* ‘ *Y*
      then obtain *x y* **where** *x* ∈ *X* **and** *y* ∈ *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 *rexp-imp-finite*, 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*, 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.

3.3 The case for *NULL*

lemma *quot-null-eq*:
  shows $(UNIV\ //\ \approx\{\}) = (\{UNIV\}::lang\ set)$
  unfolding *quotient-def Image-def str-eq-rel-def* **by** *auto*

lemma *quot-null-finiteI* [*intro*]:
  shows *finite* $((UNIV\ //\ \approx\{\})::lang\ set)$
unfolding *quot-null-eq* **by** *simp*

3.4 The case for *EMPTY*

lemma *quot-empty-subset*:
  $UNIV\ //\ (\approx\{[]\}) \subseteq \{\{[]\},\ UNIV - \{[]\}\}$
proof
  fix x
  assume $x \in UNIV\ //\ \approx\{[]\}$
  then obtain y **where** h: $x = \{z.\ (y, z) \in \approx\{[]\}\}$
    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-empty-finiteI* [*intro*]:
  shows *finite* (*UNIV* // ($\approx$\{[]\}))
by (*rule finite-subset*[*OF quot-empty-subset*]) (*simp*)

3.5 The case for *CHAR*

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

lemma *quot-char-finiteI* [*intro*]:
  shows *finite* (*UNIV*// ($\approx$\{[c]\}))
by (*rule finite-subset*[*OF quot-char-subset*]) (*simp*)

3.6 The case for *SEQ*

definition
  tag-str-SEQ :: $lang \Rightarrow lang \Rightarrow string \Rightarrow (lang \times lang\ set)$
where
  tag-str-SEQ L1 L2 $= (\lambda x.\ (\approx L1\ \text{``}\ \{x\},\ \{(\approx L2\ \text{``}\ \{x - xa\})\ |\ xa.\ \ xa \leq x \wedge xa \in L1\}))$

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 $[\![x \leq s_1; (s_1 - x) @ s_2 = y]\!] \Longrightarrow$
$\exists\ ya \leq y.\ (x @ ya) \in L_1 \wedge (y - ya) \in L_2$
using *in-seq* **by** (*rule-tac* $x = s_1 - x$ **in** *exI*, *auto elim*:*prefixE*)
moreover have $[\![s_1 \leq x; (x - s_1) @ y = s_2]\!] \Longrightarrow$
$\exists\ xa \leq x.\ xa \in L_1 \wedge (x - xa) @ y \in L_2$
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 = *tag-str-SEQ* L_1 L_2 $n \Longrightarrow m \approx(L_1 ;; L_2)\ n$
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 = *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 $\{\approx L_2\ \text{``}\ \{x - xa\}\ |xa.\ xa \leq x \wedge xa \in L_1\} =$
$\{\approx L_2\ \text{``}\ \{y - xa\}\ |xa.\ xa \leq y \wedge xa \in L_1\}$
(**is** *?Left* = *?Right*)
using *h1 tag-xy* **by** (*auto simp*:*tag-str-SEQ-def*)
moreover have $\approx L_2\ \text{``}\ \{x - xa\} \in$ *?Left* **using** *h1 h2* **by** *auto*
ultimately have $\approx L_2\ \text{``}\ \{x - xa\} \in$ *?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*: $za \leq z$ **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-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** $y @ z \in L_1 ;; L_2$
by (*drule-tac* $A = L_1$ **in** *seq-intro*, *auto elim*:*prefixE*)
}
ultimately show *?thesis* **by** *blast*
qed
} **thus** *tag-str-SEQ* L_1 L_2 m = *tag-str-SEQ* L_1 L_2 $n \Longrightarrow m \approx(L_1 ;; L_2)\ n$
by (*auto simp add*: *str-eq-def str-eq-rel-def*)
qed

lemma *quot-seq-finiteI* [*intro*]:
fixes *L1 L2*::*lang*
assumes *fin1*: *finite* (*UNIV* // ≈*L1*)
and *fin2*: *finite* (*UNIV* // ≈*L2*)
shows *finite* (*UNIV* // ≈(*L1* ;; *L2*))
proof (*rule-tac tag* = *tag-str-SEQ L1 L2* **in** *tag-finite-imageD*)
show ⋀*x y*. *tag-str-SEQ L1 L2 x* = *tag-str-SEQ L1 L2 y* ⟹ *x* ≈(*L1* ;; *L2*) *y*
by (*rule tag-str-SEQ-injI*)
next
have ∗: *finite* ((*UNIV* // ≈*L1*) × (*Pow* (*UNIV* // ≈*L2*)))
using *fin1 fin2* **by** *auto*
show *finite* (*range* (*tag-str-SEQ L1 L2*))
unfolding *tag-str-SEQ-def*
apply(*rule finite-subset*[*OF* - ∗])
unfolding *quotient-def*
by *auto*
qed

3.7 The case for *ALT*

definition
tag-str-ALT :: *lang* ⇒ *lang* ⇒ *string* ⇒ (*lang* × *lang*)
where
tag-str-ALT L1 L2 = (λ*x*. (≈*L1* ‘‘ {*x*}, ≈*L2* ‘‘ {*x*}))

lemma *quot-union-finiteI* [*intro*]:
fixes *L1 L2*::*lang*
assumes *finite1*: *finite* (*UNIV* // ≈*L1*)
and *finite2*: *finite* (*UNIV* // ≈*L2*)
shows *finite* (*UNIV* // ≈(*L1* ∪ *L2*))
proof (*rule-tac tag* = *tag-str-ALT L1 L2* **in** *tag-finite-imageD*)
show ⋀*x y*. *tag-str-ALT L1 L2 x* = *tag-str-ALT L1 L2 y* ⟹ *x* ≈(*L1* ∪ *L2*) *y*
unfolding *tag-str-ALT-def*
unfolding *str-eq-def*
unfolding *Image-def*
unfolding *str-eq-rel-def*

by *auto*
next
have $*$: *finite* $((UNIV\ //\ \approx L1) \times (UNIV\ //\ \approx L2))$
using *finite1 finite2* **by** *auto*
show *finite* (*range* (*tag-str-ALT L1 L2*))
unfolding *tag-str-ALT-def*
apply(*rule finite-subset*[*OF* - $*$])
unfolding *quotient-def*
by *auto*
qed

3.8 The case for *STAR*

This turned out to be the trickiest case.

definition
tag-str-STAR :: *lang* $\Rightarrow$ *string* $\Rightarrow$ *lang set*
where
$tag\text{-}str\text{-}STAR\ L1 = (\lambda x.\ \{\approx L1\ \text{``}\ \{x - xa\} \mid xa.\ xa < x \wedge xa \in L1\star\})$

lemma *finite-set-has-max*: $[\![finite\ A;\ A \neq \{\}]\!] \Longrightarrow$
$(\exists\ max \in A.\ \forall\ a \in A.\ f\ a <= (f\ max :: nat))$
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* $f\ a \leq f\ max$)
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-injI*:
 tag-str-STAR L_1 *m* = *tag-str-STAR* L_1 *n* $\Longrightarrow$ *m* $\approx(L_1\star)$ *n*
proof−
 { **fix** *x y z*
  assume *xz-in-star*: *x* @ *z* $\in L_1\star$
  and *tag-xy*: *tag-str-STAR* L_1 *x* = *tag-str-STAR* L_1 *y*
  have *y* @ *z* $\in L_1\star$
  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* $\in L_1\star$
    and *h3*: (*x* − *x-max*) @ *z* $\in L_1\star$
    and *h4*:$\forall$ *xa* < *x*. *xa* $\in L_1\star \wedge$ (*x* − *xa*) @ *z* $\in L_1\star$
                $\longrightarrow$ *length xa* $\leq$ *length x-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 $\exists$ *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* $\in L_1\star$ **and** *h7*: (*x* − *x-max*) $\approx L_1$ (*y* − *ya*)
   proof−
    from *tag-xy* **have** {$\approx L_1$ ‘‘ {*x* − *xa*} |*xa*. *xa* < *x* $\wedge$ *xa* $\in L_1\star$} =
     {$\approx L_1$ ‘‘ {*y* − *xa*} |*xa*. *xa* < *y* $\wedge$ *xa* $\in L_1\star$} (**is** *?left* = *?right*)
     by (*auto simp*:*tag-str-STAR-def*)
    moreover have $\approx L_1$ ‘‘ {*x* − *x-max*} $\in$ *?left* **using** *h1 h2* **by** *auto*
    ultimately have $\approx L_1$ ‘‘ {*x* − *x-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-max*) @ *z* = *a* @ *b*
     by (*drule-tac star-decom*, *auto simp*:*strict-prefix-def elim*:*prefixE*)
    have (*x* − *x-max*) $\leq$ *a* $\wedge$ (*a* − (*x* − *x-max*)) @ *b* = *z*

```
      proof -
        have ((x - x-max) ≤ a ∧ (a - (x - x-max)) @ b = z) ∨
                       (a < (x - x-max) ∧ ((x - x-max) - a) @ z = b)
          using app-eq-dest[OF ab-max] by (auto simp:strict-prefix-def)
        moreover {
          assume np: a < (x - x-max) and b-eqs: ((x - x-max) - a) @ z = b
          have False
          proof -
            let ?x-max' = x-max @ a
            have ?x-max' < x
              using np h1 by (clarsimp simp:strict-prefix-def diff-prefix)
            moreover have ?x-max' ∈ L_1⋆
              using a-in h2 by (simp add:star-intro3)
            moreover have (x - ?x-max') @ z ∈ L_1⋆
              using b-eqs 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 ∈ L_1
        using a-in by (auto elim:prefixE)
      from x-za h7 have (y - ya) @ za ∈ 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 = tag-str-STAR L_1 n ⟹ m ≈(L_1⋆) n
  by (auto simp add:str-eq-def str-eq-rel-def)
qed


lemma quot-star-finiteI [intro]:
  fixes L1::lang
  assumes finite1: finite (UNIV // ≈L1)
  shows finite (UNIV // ≈(L1⋆))
proof (rule-tac tag = tag-str-STAR L1 in tag-finite-imageD)
  show ⋀x y. tag-str-STAR L1 x = tag-str-STAR L1 y ⟹ x ≈(L1⋆) y
    by (rule tag-str-STAR-injI)
next
  have *: finite (Pow (UNIV // ≈L1))
    using finite1 by auto
  show finite (range (tag-str-STAR L1))
    unfolding tag-str-STAR-def
    apply(rule finite-subset[OF - *])
```

  unfolding *quotient-def*
  by *auto*
qed

3.9 The main lemma

lemma *rexp-imp-finite*:
 fixes r::*rexp*
 shows *finite* (*UNIV* // $\approx$(L r))
by (*induct* r) (*auto*)

end