

tphols-2011

By xingyuan

January 30, 2011

Contents

1	List prefixes and postfixes	1
1.1	Prefix order on lists	1
1.2	Basic properties of prefixes	2
1.3	Parallel lists	5
1.4	Postfix order on lists	6
2	A small theory of prefix subtraction	9
3	Preliminary definitions	10
4	A slightly modified version of Arden's lemma	13
5	Direction <i>finite partition</i> \Rightarrow <i>regular language</i>	16
5.1	The proof of this direction	21
5.1.1	Basic properties	21
5.1.2	Intialization	22
5.1.3	Iteration step	24
5.1.4	Conclusion of the proof	30
6	Direction <i>regular language</i> \Rightarrow <i>finite partition</i>	33
6.1	The scheme	33
6.2	The proof	37
6.2.1	The case for <i>NULL</i>	37
6.2.2	The case for <i>EMPTY</i>	37
6.2.3	The case for <i>CHAR</i>	38
6.2.4	The case for <i>SEQ</i>	39
6.2.5	The case for <i>ALT</i>	40
6.2.6	The case for <i>STAR</i>	41
6.2.7	The conclusion	49

1 List prefixes and postfixes

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

1.1 Prefix order on lists

```
instantiation list :: (type) {order, bot}
begin
```

definition

prefix-def: $xs \leq ys \longleftrightarrow (\exists zs. ys = xs @ zs)$

definition

strict-prefix-def: $xs < ys \longleftrightarrow xs \leq ys \wedge xs \neq (ys::'a \text{ list})$

definition

bot = []

instance proof

qed (*auto simp add: prefix-def strict-prefix-def bot-list-def*)

end

lemma *prefixI* [*intro?*]: $ys = xs @ zs \implies xs \leq ys$
unfolding *prefix-def* **by** *blast*

lemma *prefixE* [*elim?*]:

assumes $xs \leq ys$

obtains zs **where** $ys = xs @ zs$

using *assms* **unfolding** *prefix-def* **by** *blast*

lemma *strict-prefixI'* [*intro?*]: $ys = xs @ z \# zs \implies xs < ys$
unfolding *strict-prefix-def prefix-def* **by** *blast*

lemma *strict-prefixE'* [*elim?*]:

assumes $xs < ys$

obtains $z \ zs$ **where** $ys = xs @ z \# zs$

proof –

from $\langle xs < ys \rangle$ **obtain** us **where** $ys = xs @ us$ **and** $xs \neq ys$

unfolding *strict-prefix-def prefix-def* **by** *blast*

with that show *?thesis* **by** (*auto simp add: neq-Nil-conv*)

qed

lemma *strict-prefixI* [*intro?*]: $xs \leq ys \implies xs \neq ys \implies xs < (ys::'a \text{ list})$
unfolding *strict-prefix-def* **by** *blast*

lemma *strict-prefixE* [*elim?*]:

fixes $xs \ ys :: 'a \text{ list}$

assumes $xs < ys$
obtains $xs \leq ys$ **and** $xs \neq ys$
using *assms* **unfolding** *strict-prefix-def* **by** *blast*

1.2 Basic properties of prefixes

theorem *Nil-prefix [iff]*: $[] \leq xs$
by (*simp add: prefix-def*)

theorem *prefix-Nil [simp]*: $(xs \leq []) = (xs = [])$
by (*induct xs*) (*simp-all add: prefix-def*)

lemma *prefix-snoc [simp]*: $(xs \leq ys @ [y]) = (xs = ys @ [y] \vee xs \leq ys)$

proof

assume $xs \leq ys @ [y]$

then obtain zs **where** $zs @ [y] = xs @ zs$..

show $xs = ys @ [y] \vee xs \leq ys$

by (*metis append-Nil2 butlast-append butlast-snoc prefixI zs*)

next

assume $xs = ys @ [y] \vee xs \leq ys$

then show $xs \leq ys @ [y]$

by (*metis order-eq-iff strict-prefixE strict-prefixI' xt1 (7)*)

qed

lemma *Cons-prefix-Cons [simp]*: $(x \# xs \leq y \# ys) = (x = y \wedge xs \leq ys)$
by (*auto simp add: prefix-def*)

lemma *less-eq-list-code [code]*:

$([]::'a::\{equal, ord\} list) \leq xs \longleftrightarrow True$

$(x::'a::\{equal, ord\}) \# xs \leq [] \longleftrightarrow False$

$(x::'a::\{equal, ord\}) \# xs \leq y \# ys \longleftrightarrow x = y \wedge xs \leq ys$

by *simp-all*

lemma *same-prefix-prefix [simp]*: $(xs @ ys \leq xs @ zs) = (ys \leq zs)$
by (*induct xs*) *simp-all*

lemma *same-prefix-nil [iff]*: $(xs @ ys \leq xs) = (ys = [])$
by (*metis append-Nil2 append-self-conv order-eq-iff prefixI*)

lemma *prefix-prefix [simp]*: $xs \leq ys \implies xs \leq ys @ zs$
by (*metis order-le-less-trans prefixI strict-prefixE strict-prefixI*)

lemma *append-prefixD*: $xs @ ys \leq zs \implies xs \leq zs$
by (*auto simp add: prefix-def*)

theorem *prefix-Cons*: $(xs \leq y \# ys) = (xs = [] \vee (\exists zs. xs = y \# zs \wedge zs \leq ys))$
by (*cases xs*) (*auto simp add: prefix-def*)

theorem *prefix-append*:

$(xs \leq ys @ zs) = (xs \leq ys \vee (\exists us. xs = ys @ us \wedge us \leq zs))$
apply (*induct zs rule: rev-induct*)
apply *force*
apply (*simp del: append-assoc add: append-assoc [symmetric]*)
apply (*metis append-eq-appendI*)
done

lemma *append-one-prefix*:
 $xs \leq ys \implies \text{length } xs < \text{length } ys \implies xs @ [ys ! \text{length } xs] \leq ys$
unfolding *prefix-def*
by (*metis Cons-eq-appendI append-eq-appendI append-eq-conv-conj eq-Nil-appendI nth-drop'*)

theorem *prefix-length-le*: $xs \leq ys \implies \text{length } xs \leq \text{length } ys$
by (*auto simp add: prefix-def*)

lemma *prefix-same-cases*:
 $(xs_1 :: 'a \text{ list}) \leq ys \implies xs_2 \leq ys \implies xs_1 \leq xs_2 \vee xs_2 \leq xs_1$
unfolding *prefix-def* **by** (*metis append-eq-append-conv2*)

lemma *set-mono-prefix*: $xs \leq ys \implies \text{set } xs \subseteq \text{set } ys$
by (*auto simp add: prefix-def*)

lemma *take-is-prefix*: $\text{take } n \text{ } xs \leq xs$
unfolding *prefix-def* **by** (*metis append-take-drop-id*)

lemma *map-prefixI*: $xs \leq ys \implies \text{map } f \text{ } xs \leq \text{map } f \text{ } ys$
by (*auto simp: prefix-def*)

lemma *prefix-length-less*: $xs < ys \implies \text{length } xs < \text{length } ys$
by (*auto simp: strict-prefix-def prefix-def*)

lemma *strict-prefix-simps* [*simp, code*]:
 $xs < [] \longleftrightarrow \text{False}$
 $[] < x \# xs \longleftrightarrow \text{True}$
 $x \# xs < y \# ys \longleftrightarrow x = y \wedge xs < ys$
by (*simp-all add: strict-prefix-def cong: conj-cong*)

lemma *take-strict-prefix*: $xs < ys \implies \text{take } n \text{ } xs < ys$
apply (*induct n arbitrary: xs ys*)
apply (*case-tac ys, simp-all*)[1]
apply (*metis order-less-trans strict-prefixI take-is-prefix*)
done

lemma *not-prefix-cases*:
assumes *pf*: $\neg ps \leq ls$
obtains
 $(c1) \text{ } ps \neq [] \text{ and } ls = []$
 $| (c2) \text{ } a \text{ as } xs \text{ where } ps = a \# as \text{ and } ls = x \# xs \text{ and } x = a \text{ and } \neg as \leq xs$

```

| (cβ) a as x xs where ps = a#as and ls = x#xs and x ≠ a
proof (cases ps)
  case Nil then show ?thesis using pfx by simp
next
  case (Cons a as)
  note c = ⟨ps = a#as⟩
  show ?thesis
  proof (cases ls)
    case Nil then show ?thesis by (metis append-Nil2 pfx c1 same-prefix-nil)
  next
    case (Cons x xs)
    show ?thesis
    proof (cases x = a)
      case True
      have ¬ as ≤ xs using pfx c Cons True by simp
      with c Cons True show ?thesis by (rule c2)
    next
      case False
      with c Cons show ?thesis by (rule c3)
    qed
  qed
qed

```

```

lemma not-prefix-induct [consumes 1, case-names Nil Neq Eq]:
  assumes np: ¬ ps ≤ ls
    and base: ∧x xs. P (x#xs) []
    and r1: ∧x xs y ys. x ≠ y ⇒ P (x#xs) (y#ys)
    and r2: ∧x xs y ys. [ x = y; ¬ xs ≤ ys; P xs ys ] ⇒ P (x#xs) (y#ys)
  shows P ps ls using np
proof (induct ls arbitrary: ps)
  case Nil then show ?case
    by (auto simp: neq-Nil-conv elim!: not-prefix-cases intro!: base)
next
  case (Cons y ys)
  then have npfx: ¬ ps ≤ (y # ys) by simp
  then obtain x xs where pv: ps = x # xs
    by (rule not-prefix-cases) auto
  show ?case by (metis Cons.hyps Cons-prefix-Cons npfx pv r1 r2)
qed

```

1.3 Parallel lists

definition

```

parallel :: 'a list => 'a list => bool (infixl || 50) where
(x || ys) = (¬ xs ≤ ys ∧ ¬ ys ≤ xs)

```

```

lemma parallelI [intro]: ¬ xs ≤ ys ==> ¬ ys ≤ xs ==> xs || ys
unfolding parallel-def by blast

```

```

lemma parallelE [elim]:
  assumes  $xs \parallel ys$ 
  obtains  $\neg xs \leq ys \wedge \neg ys \leq xs$ 
  using assms unfolding parallel-def by blast

theorem prefix-cases:
  obtains  $xs \leq ys \mid ys < xs \mid xs \parallel ys$ 
  unfolding parallel-def strict-prefix-def by blast

theorem parallel-decomp:
   $xs \parallel ys \implies \exists as\ b\ bs\ c\ cs. b \neq c \wedge xs = as @ b \# bs \wedge ys = as @ c \# cs$ 
proof (induct xs rule: rev-induct)
  case Nil
  then have False by auto
  then show ?case ..
next
  case (snoc x xs)
  show ?case
  proof (rule prefix-cases)
    assume le: xs ≤ ys
    then obtain ys' where  $ys = xs @ ys' ..$ 
    show ?thesis
    proof (cases ys')
      assume  $ys' = []$ 
      then show ?thesis by (metis append-Nil2 parallelE prefixI snoc.premys ys)
    next
      fix c cs assume  $ys': ys' = c \# cs$ 
      then show ?thesis
      by (metis Cons-eq-appendI eq-Nil-appendI parallelE prefixI same-prefix-prefix snoc.premys ys)
    qed
  next
    assume  $ys < xs$  then have  $ys \leq xs @ [x]$  by (simp add: strict-prefix-def)
    with snoc have False by blast
    then show ?thesis ..
  next
    assume  $xs \parallel ys$ 
    with snoc obtain as b bs c cs where  $neg: (b::'a) \neq c$ 
      and  $xs: xs = as @ b \# bs$  and  $ys: ys = as @ c \# cs$ 
      by blast
    from xs have  $xs @ [x] = as @ b \# (bs @ [x])$  by simp
    with neg ys show ?thesis by blast
  qed
qed

lemma parallel-append:  $a \parallel b \implies a @ c \parallel b @ d$ 
apply (rule parallelI)
apply (erule parallelE, erule conjE,
  induct rule: not-prefix-induct, simp+)

```

done

lemma *parallel-appendI*: $xs \parallel ys \implies x = xs @ xs' \implies y = ys @ ys' \implies x \parallel y$
by (*simp add: parallel-append*)

lemma *parallel-commute*: $a \parallel b \longleftrightarrow b \parallel a$
unfolding *parallel-def* by *auto*

1.4 Postfix order on lists

definition

postfix :: 'a list => 'a list => bool ((-/ >>= -) [51, 50] 50) **where**
(*xs >>= ys*) = ($\exists zs. xs = zs @ ys$)

lemma *postfixI* [*intro?*]: $xs = zs @ ys \implies xs \gg= ys$
unfolding *postfix-def* by *blast*

lemma *postfixE* [*elim?*]:
assumes $xs \gg= ys$
obtains *zs* **where** $xs = zs @ ys$
using *assms* unfolding *postfix-def* by *blast*

lemma *postfix-refl* [*iff*]: $xs \gg= xs$
by (*auto simp add: postfix-def*)

lemma *postfix-trans*: $\llbracket xs \gg= ys; ys \gg= zs \rrbracket \implies xs \gg= zs$
by (*auto simp add: postfix-def*)

lemma *postfix-antisym*: $\llbracket xs \gg= ys; ys \gg= xs \rrbracket \implies xs = ys$
by (*auto simp add: postfix-def*)

lemma *Nil-postfix* [*iff*]: $xs \gg= []$
by (*simp add: postfix-def*)

lemma *postfix-Nil* [*simp*]: $([] \gg= xs) = (xs = [])$
by (*auto simp add: postfix-def*)

lemma *postfix-ConsI*: $xs \gg= ys \implies x\#xs \gg= ys$
by (*auto simp add: postfix-def*)

lemma *postfix-ConsD*: $xs \gg= y\#ys \implies xs \gg= ys$
by (*auto simp add: postfix-def*)

lemma *postfix-appendI*: $xs \gg= ys \implies zs @ xs \gg= ys$
by (*auto simp add: postfix-def*)

lemma *postfix-appendD*: $xs \gg= zs @ ys \implies xs \gg= ys$
by (*auto simp add: postfix-def*)

lemma *postfix-is-subset*: $xs \gg= ys \implies \text{set } ys \subseteq \text{set } xs$
proof –

assume $xs \gg= ys$
then obtain *zs* **where** $xs = zs @ ys$..
then show *?thesis* by (*induct zs*) *auto*

qed

lemma postfix-ConsD2: $x\#xs \gg= y\#ys \implies xs \gg= ys$

proof –

assume $x\#xs \gg= y\#ys$

then obtain zs **where** $x\#xs = zs @ y\#ys$..

then show *?thesis*

by (*induct zs*) (*auto intro!: postfix-appendI postfix-ConsI*)

qed

lemma postfix-to-prefix [code]: $xs \gg= ys \iff rev\ ys \leq rev\ xs$

proof

assume $xs \gg= ys$

then obtain zs **where** $xs = zs @ ys$..

then have $rev\ xs = rev\ ys @ rev\ zs$ **by** *simp*

then show $rev\ ys \leq rev\ xs$..

next

assume $rev\ ys \leq rev\ xs$

then obtain zs **where** $rev\ xs = rev\ ys @ zs$..

then have $rev\ (rev\ xs) = rev\ zs @ rev\ (rev\ ys)$ **by** *simp*

then have $xs = rev\ zs @ ys$ **by** *simp*

then show $xs \gg= ys$..

qed

lemma distinct-postfix: $distinct\ xs \implies xs \gg= ys \implies distinct\ ys$

by (*clarsimp elim!: postfixE*)

lemma postfix-map: $xs \gg= ys \implies map\ f\ xs \gg= map\ f\ ys$

by (*auto elim!: postfixE intro: postfixI*)

lemma postfix-drop: $as \gg= drop\ n\ as$

unfolding *postfix-def*

apply (*rule exI [where x = take n as]*)

apply *simp*

done

lemma postfix-take: $xs \gg= ys \implies xs = take\ (length\ xs - length\ ys)\ xs @ ys$

by (*clarsimp elim!: postfixE*)

lemma parallelD1: $x \parallel y \implies \neg x \leq y$

by *blast*

lemma parallelD2: $x \parallel y \implies \neg y \leq x$

by *blast*

lemma parallel-Nil1 [simp]: $\neg x \parallel []$

unfolding *parallel-def* **by** *simp*

lemma parallel-Nil2 [simp]: $\neg [] \parallel x$


```

unfolding parallel-def by simp

lemma Cons-parallelI1:  $a \neq b \implies a \# as \parallel b \# bs$ 
by auto

lemma Cons-parallelI2:  $\llbracket a = b; as \parallel bs \rrbracket \implies a \# as \parallel b \# bs$ 
by (metis Cons-prefix-Cons parallelE parallelI)

lemma not-equal-is-parallel:
  assumes neq:  $xs \neq ys$ 
  and len:  $length\ xs = length\ ys$ 
  shows  $xs \parallel ys$ 
  using len neq
proof (induct rule: list-induct2)
  case Nil
  then show ?case by simp
next
  case (Cons a as b bs)
  have ih:  $as \neq bs \implies as \parallel bs$  by fact
  show ?case
  proof (cases a = b)
  case True
  then have  $as \neq bs$  using Cons by simp
  then show ?thesis by (rule Cons-parallelI2 [OF True ih])
  next
  case False
  then show ?thesis by (rule Cons-parallelI1)
qed
qed

end

theory Prefix-subtract
  imports Main List-Prefix
begin

```

2 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 *Prelude*
imports *Main*
begin

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

by *blast*

end
theory *Myhill-1*
 imports *Main List-Prefix Prefix-subtract Prelude*
begin

3 Preliminary definitions

types *lang* = *string set*

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

definition *Seq* :: *lang* \Rightarrow *lang* \Rightarrow *lang* (**infixr** ;; 100)

where

$A ;; B = \{s_1 @ s_2 \mid s_1 s_2. s_1 \in A \wedge s_2 \in B\}$

Transitive closure of language *L*.

inductive-set

Star :: *lang* \Rightarrow *lang* (**-*** [101] 102)

for *L*

where

start[*intro*]: $\square \in L^\star$

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

Some properties of operator ;;.

lemma *seq-union-distrib-right*:

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

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

lemma *seq-union-distrib-left*:

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

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

lemma *seq-intro*:

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

by (*auto simp:Seq-def*)

lemma *seq-assoc*:

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

unfolding *Seq-def*

apply(*auto*)

apply(*blast*)

by (*metis append-assoc*)

lemma *seq-empty* [*simp*]:

shows $A ;; \{\square\} = A$

and $\{\square\} ;; A = A$

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

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 *lang-star-cases*:

shows $L\star = \{[]\} \cup L ;; L\star$

proof

{ **fix** x

have $x \in L\star \implies x \in \{[]\} \cup L ;; L\star$

unfolding *Seq-def*

by (*induct rule: Star.induct*) (*auto*)

}

then show $L\star \subseteq \{[]\} \cup L ;; L\star$ **by** *auto*

next

show $\{[]\} \cup L ;; L\star \subseteq L\star$

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

qed

fun

$\text{pow} :: \text{lang} \Rightarrow \text{nat} \Rightarrow \text{lang}$ (**infixl** \uparrow 100)

where

$A \uparrow 0 = \{[]\}$

| $A \uparrow (\text{Suc } n) = A ;; (A \uparrow n)$

lemma *star-pow-eq*:

shows $A\star = (\bigcup n. A \uparrow n)$

proof –

{ **fix** $n x$

assume $x \in (A \uparrow n)$

then have $x \in A\star$

by (*induct n arbitrary: x*) (*auto simp add: Seq-def*)

}

moreover

{ **fix** x

assume $x \in A\star$

```

then have  $\exists n. x \in A \uparrow n$ 
proof (induct rule: Star.induct)
  case start
  have  $\square \in A \uparrow 0$  by auto
  then show  $\exists n. \square \in A \uparrow n$  by blast
next
  case (step s1 s2)
  have  $s1 \in A$  by fact
  moreover
  have  $\exists n. s2 \in A \uparrow n$  by fact
  then obtain  $n$  where  $s2 \in A \uparrow n$  by blast
  ultimately
  have  $s1 @ s2 \in A \uparrow (\text{Suc } n)$  by (auto simp add: Seq-def)
  then show  $\exists n. s1 @ s2 \in A \uparrow n$  by blast
qed
}
ultimately show  $A^\star = (\bigcup n. A \uparrow n)$  by auto
qed

```

```

lemma
  shows seq-Union-left:  $B ;; (\bigcup n. A \uparrow n) = (\bigcup n. B ;; (A \uparrow n))$ 
  and seq-Union-right:  $(\bigcup n. A \uparrow n) ;; B = (\bigcup n. (A \uparrow n) ;; B)$ 
unfolding Seq-def by auto

```

```

lemma seq-pow-comm:
  shows  $A ;; (A \uparrow n) = (A \uparrow n) ;; A$ 
by (induct n) (simp-all add: seq-assoc[symmetric])

```

```

lemma seq-star-comm:
  shows  $A ;; A^\star = A^\star ;; A$ 
unfolding star-pow-eq
unfolding seq-Union-left
unfolding seq-pow-comm
unfolding seq-Union-right
by simp

```

Two lemmas about the length of strings in $A \uparrow n$

```

lemma pow-length:
  assumes  $a: \square \notin A$ 
  and  $b: s \in A \uparrow \text{Suc } n$ 
  shows  $n < \text{length } s$ 
using b
proof (induct n arbitrary: s)
  case 0
  have  $s \in A \uparrow \text{Suc } 0$  by fact
  with a have  $s \neq \square$  by auto
  then show  $0 < \text{length } s$  by auto
next
  case (Suc n)

```

have $ih: \bigwedge s. s \in A \uparrow \text{Suc } n \implies n < \text{length } s$ **by fact**
have $s \in A \uparrow \text{Suc } (\text{Suc } n)$ **by fact**
then obtain $s1\ s2$ **where** $eq: s = s1 @ s2$ **and** $*$: $s1 \in A$ **and** $**$: $s2 \in A \uparrow \text{Suc } n$
by (*auto simp add: Seq-def*)
from $ih\ **$ **have** $n < \text{length } s2$ **by simp**
moreover have $0 < \text{length } s1$ **using** $*$ **by auto**
ultimately show $\text{Suc } n < \text{length } s$ **unfolding eq**
by (*simp only: length-append*)
qed

lemma seq-pow-length:
assumes $a: [] \notin A$
and $b: s \in B ;; (A \uparrow \text{Suc } n)$
shows $n < \text{length } s$
proof –
from b **obtain** $s1\ s2$ **where** $eq: s = s1 @ s2$ **and** $*$: $s2 \in A \uparrow \text{Suc } n$
unfolding Seq-def **by auto**
from $*$ **have** $n < \text{length } s2$ **by** (*rule pow-length[OF a]*)
then show $n < \text{length } s$ **using eq** **by simp**
qed

4 A slightly modified version of Arden's lemma

Arden's lemma expressed at the level of languages, rather than the level of regular expression.

lemma ardens-helper:
assumes $eq: X = X ;; A \cup B$
shows $X = X ;; (A \uparrow \text{Suc } n) \cup (\bigcup_{m \in \{0..n\}} B ;; (A \uparrow m))$
proof (*induct n*)
case 0
show $X = X ;; (A \uparrow \text{Suc } 0) \cup (\bigcup_{(m::\text{nat}) \in \{0..0\}} B ;; (A \uparrow m))$
using eq **by simp**
next
case ($\text{Suc } n$)
have $ih: X = X ;; (A \uparrow \text{Suc } n) \cup (\bigcup_{m \in \{0..n\}} B ;; (A \uparrow m))$ **by fact**
also have $\dots = (X ;; A \cup B) ;; (A \uparrow \text{Suc } n) \cup (\bigcup_{m \in \{0..n\}} B ;; (A \uparrow m))$
using eq **by simp**
also have $\dots = X ;; (A \uparrow \text{Suc } (\text{Suc } n)) \cup (B ;; (A \uparrow \text{Suc } n)) \cup (\bigcup_{m \in \{0..n\}} B ;; (A \uparrow m))$
by (*simp add: seq-union-distrib-right seq-assoc*)
also have $\dots = X ;; (A \uparrow \text{Suc } (\text{Suc } n)) \cup (\bigcup_{m \in \{0..\text{Suc } n\}} B ;; (A \uparrow m))$
by (*auto simp add: le-Suc-eq*)
finally show $X = X ;; (A \uparrow \text{Suc } (\text{Suc } n)) \cup (\bigcup_{m \in \{0..\text{Suc } n\}} B ;; (A \uparrow m))$.
qed

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  $A^\star = \{\emptyset\} \cup A^\star ;; A$ 
    unfolding seq-star-comm[symmetric]
    by (rule lang-star-cases)
  then have  $B ;; A^\star = B ;; (\{\emptyset\} \cup A^\star ;; A)$ 
    unfolding Seq-def by simp
  also have  $\dots = B \cup B ;; (A^\star ;; A)$ 
    unfolding seq-union-distrib-left by simp
  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:  $X = X ;; A \cup B$ 
  { fix n::nat
    have  $B ;; (A \uparrow n) \subseteq X$  using ardens-helper[OF eq, of n] by auto }
  then have  $B ;; A^\star \subseteq X$  unfolding star-pow-eq Seq-def
    by (auto simp add: UNION-def)
  moreover
  { fix s::string
    obtain k where  $k = \text{length } s$  by auto
    then have not-in:  $s \notin X ;; (A \uparrow \text{Suc } k)$ 
      using seq-pow-length[OF nemp] by blast
    assume  $s \in X$ 
    then have  $s \in X ;; (A \uparrow \text{Suc } k) \cup (\bigcup_{m \in \{0..k\}} B ;; (A \uparrow m))$ 
      using ardens-helper[OF eq, of k] by auto
    then have  $s \in (\bigcup_{m \in \{0..k\}} B ;; (A \uparrow m))$  using not-in by auto
    moreover
    have  $(\bigcup_{m \in \{0..k\}} B ;; (A \uparrow m)) \subseteq (\bigcup_n B ;; (A \uparrow n))$  by auto
    ultimately
    have  $s \in B ;; A^\star$  unfolding star-pow-eq seq-Union-left
      by auto }
  then have  $X \subseteq B ;; A^\star$  by auto
  ultimately
  show  $X = B ;; A^\star$  by simp
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 $L :: 'a \Rightarrow \text{string set}$

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

overloading $L\text{-rexp} \equiv L :: \text{rexp} \Rightarrow \text{string set}$

begin

fun

$L\text{-rexp} :: \text{rexp} \Rightarrow \text{string set}$

where

$L\text{-rexp} (\text{NULL}) = \{\}$

| $L\text{-rexp} (\text{EMPTY}) = \{\}\}$

| $L\text{-rexp} (\text{CHAR } c) = \{[c]\}$

| $L\text{-rexp} (\text{SEQ } r1\ r2) = (L\text{-rexp } r1) ;; (L\text{-rexp } r2)$

| $L\text{-rexp} (\text{ALT } r1\ r2) = (L\text{-rexp } r1) \cup (L\text{-rexp } r2)$

| $L\text{-rexp} (\text{STAR } r) = (L\text{-rexp } r)^\star$

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 $\text{folds } f$ makes sense when f is not *associative* and *commutitive*, while $\text{fold } f$ does not.

definition

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

where

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

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

lemma folds-alt-simp [simp]:

$\text{finite } rs \implies L(\text{folds } \text{ALT } \text{NULL } rs) = \bigcup (L \text{ ` } rs)$

apply (rule set-eq-intro , $\text{simp add:folds-def}$)

apply (rule someI2-ex , $\text{erule finite-imp-fold-graph}$)

by ($\text{erule fold-graph.induct}$, 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

$\text{str-eq-rel } (\approx - [100] 100)$

where

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

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

definition

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

The following lemma show the relationship between $\mathit{finals}(\text{Lang})$ and Lang .

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

$$\text{Lang} = \bigcup \mathit{finals}(\text{Lang})$$

proof

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

proof

fix x

assume $x \in \text{Lang}$

thus $x \in \bigcup (\mathit{finals} \text{ Lang})$

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

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

qed

next

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

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

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

qed

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

The relationship between equivalent classes can be described by an equational system. For example, in equational system (1), X_0, X_1 are equivalent classes. The first equation says every string in X_0 is obtained either by appending one b to a string in X_0 or by appending one a to a string in X_1 or just be an empty string (represented by the regular expression 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 datatype *rhs-item*, mnemonic for 'right hand side item'. Generally, there are two kinds of right hand side items, one kind corresponds to pure regular expressions, like the λ in (1), the other kind corresponds to transitions from one one equivalent class to another, like the X_0b, X_1a etc.

datatype *rhs-item* =

Lam *rexp*

| *Trn* (*string set*) *rexp*

In this formalization, pure regular expressions like 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
 $\{Lam(EMPTY)\} \cup \{Trn Y (CHAR c) \mid Y c. Y \in CS \wedge Y ;; \{[c]\} \subseteq X\}$
 else
 $\{Trn Y (CHAR c) \mid Y c. Y \in CS \wedge Y ;; \{[c]\} \subseteq X\}$

In the definition of *init-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 describing the formation of every equivalent class inside *CS* is given by the following *eqs(CS)*.

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

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

definition
items-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

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

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

definition

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

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 the regular expression corresponds to *rhs*.

definition

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

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

fun *attach-rexp* :: *rexp* \Rightarrow *rhs-item* \Rightarrow *rhs-item*

where

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

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

definition

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

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

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

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

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

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

definition

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

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

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*]:

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

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

show *?thesis*

proof(*rule h*)

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

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-eqvas $ES \equiv$

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

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

definition

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

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

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

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) occurring 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

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

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

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

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

definition

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

5.1 The proof of this direction

5.1.1 Basic properties

The following are some basic properties of the above definitions.

lemma *L-rhs-union-distrib*:

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

by *simp*

lemma *finite-snd-Trn*:

assumes *finite:finite rhs*
shows *finite {r₂. Trn Y r₂ ∈ rhs}* (**is** *finite ?B*)
proof –
def *rhs'* ≡ {*e* ∈ *rhs*. ∃ *r*. *e* = *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 $\square \notin L$ (*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 (*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 (*items-of rhs X*) = X ;; (L (*rexp-of rhs X*))
proof –
have *finite* ((*snd* *o* *the-Trn*) ‘*items-of rhs X*) **using** *finite-items-of[OF finite]*
by *auto*
thus *?thesis*
apply (*auto simp:rexp-of-def Seq-def items-of-def*)
apply (*rule-tac x = s₁ in exI, rule-tac x = s₂ in exI, auto*)
by (*rule-tac x = Trn X r in exI, auto simp:Seq-def*)
qed

lemma *rexp-of-lam-eq-lam-set*:
assumes *finite:finite rhs*
shows L (*rexp-of-lam rhs*) = L (*lam-of rhs*)
proof –
have *finite* (*the-r* ‘ {*Lam r* | *r*. *Lam r* ∈ *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*)
apply (*rule-tac x = s₁ @ s₁' in exI, rule-tac x = s₂' in exI*)
apply (*auto simp:Seq-def*)

done

lemma *lang-of-append-rhs*:

$L (\text{append-rhs-rexp } rhs \ r) = L \ rhs \ ; ; \ L \ r$
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*:

$\text{classes-of } A \cup \text{classes-of } B = \text{classes-of } (A \cup B)$
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*)

5.1.2 Intialization

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

lemma *defined-by-str*:

$\llbracket s \in X; X \in UNIV // (\approx Lang) \rrbracket \implies X = (\approx Lang) \ \{s\}$
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 $Y \equiv (\approx Lang) \ \{s\}$
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-egs*:

assumes *X-in-egs*: $(X, xrhs) \in (\text{egs } (UNIV // (\approx Lang)))$
shows $X = L \ xrhs$

```

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

lemma finite-init-rhs:
  assumes  $\text{finite}: \text{finite } CS$ 
  shows  $\text{finite} (\text{init-rhs } CS X)$ 
proof –
  have  $\text{finite} \{ \text{Trn } Y (\text{CHAR } c) \mid Y c. Y \in CS \wedge Y ;; \{[c]\} \subseteq X \}$  (is  $\text{finite } ?A$ )
proof –
  def  $S \equiv \{ (Y, c) \mid Y c. Y \in CS \wedge Y ;; \{[c]\} \subseteq X \}$ 
  def  $h \equiv \lambda (Y, c). \text{Trn } Y (\text{CHAR } c)$ 
  have  $\text{finite} (CS \times (\text{UNIV}::\text{char set}))$  using  $\text{finite}$  by auto
  hence  $\text{finite } S$  using  $S\text{-def}$ 
  by (rule-tac  $B = CS \times \text{UNIV}$  in  $\text{finite-subset}$ , auto)
  moreover have  $?A = h ' S$  by (auto simp:  $S\text{-def } h\text{-def image-def}$ )
  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-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

5.1.3 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*★

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-left)
ultimately show *?thesis* **by** *simp*
qed

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

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

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

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

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

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

lemma *rhs-subst-keeps-nonempty*:
 $\llbracket \text{rhs-nonempty rhs}; \text{rhs-nonempty xrhs} \rrbracket \implies \text{rhs-nonempty (rhs-subst rhs X xrhs)}$
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 $L(\text{rhs-subst rhs X xrhs}) = L \text{ rhs}$ (**is** $?Left = ?Right$)
proof –
def $A \equiv L(\text{rhs} - \text{items-of rhs X})$
have $?Left = A \cup L(\text{append-rhs-rexp xrhs (rexp-of rhs X)})$
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 $\text{rhs} = (\text{rhs} - \text{items-of rhs X}) \cup (\text{items-of rhs X})$ **by** (*auto simp: items-of-def*)
thus *?thesis* **by** (*simp only: L-rhs-union-distrib A-def*)
qed

moreover have $L(\text{append-rhs-rexp } xrhs (\text{rexp-of rhs } X)) = L(\text{items-of rhs } X)$
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*:
 $\llbracket \text{finite rhs}; \text{finite yrhs} \rrbracket \implies \text{finite } (\text{rhs-subst rhs } Y \text{ yrhs})$
by (*auto simp:rhs-subst-def append-keeps-finite*)

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

proof–
def *eqns'* $\equiv \{((Ya :: \text{string set}), \text{yrhsa}) \mid Ya \text{ yrhsa. } (Ya, \text{yrhsa}) \in ES\}$
def *h* $\equiv \lambda ((Ya :: \text{string set}), \text{yrhsa}). (Ya, \text{rhs-subst yrhsa } Y \text{ yrhs})$
have *finite* (*h* ‘*eqns'*) **using** *finite h-def eqns'-def* **by** *auto*
moreover have *?A* = *h* ‘*eqns'* **by** (*auto simp:h-def eqns'-def*)
ultimately show *?thesis* **by** *auto*

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

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

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

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

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

lemma *rhs-subst-updates-cls*:
 $X \notin \text{classes-of } xrhs \implies$
 $\text{classes-of } (\text{rhs-subst rhs } X \text{ } xrhs) = \text{classes-of rhs} \cup \text{classes-of } xrhs - \{X\}$
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 ($ES \cup \{(Y, yrhs)\}$) (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 $xrhs-xrhs'$: $xrhs' = rhs\text{-subst } xrhs \ Y \ (arden\text{-variate } Y \ yrhs)$

and $X\text{-in}$: $(X, xrhs) \in ES$ 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 $X\text{-in}$ 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 $Inv\text{-ES}$: Inv ($ES \cup \{(Y, yrhs)\}$)

shows Inv (eqs-subst ES Y (arden-variate Y $yrhs$))

proof –

have finite-yrhs: finite $yrhs$

using $Inv\text{-ES}$ by (auto simp:Inv-def finite-rhs-def)

have nonempty-yrhs: rhs-nonempty $yrhs$

using $Inv\text{-ES}$ by (auto simp:Inv-def ardenable-def)

have $Y\text{-eq-yrhs}$: $Y = L$ $yrhs$

using $Inv\text{-ES}$ by (simp only:Inv-def valid-eqns-def, blast)

have distinct-eqns (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 ≡ λ 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 ⇒ ∃ xrhs'. (X, xrhs') ∈ (eqs-subst ES Y yrhs)
by (auto simp:eqs-subst-def)

lemma card-noteq-1-has-more:
  assumes card:card S ≠ 1
  and e-in: e ∈ S
  and finite: finite S
  obtains e' where e' ∈ S ∧ e ≠ e'
proof –
  have card (S – {e}) > 0
  proof –
    have 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} ≠ {} using finite by (rule-tac notI, simp)
  thus (∧ e'. e' ∈ S ∧ e ≠ e' ⇒ thesis) ⇒ 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 $(\text{card } ?ES'', \text{card } ES) \in \text{less-than}$
proof –
have *finite* ES' **using** *finite-ES* ES' -def **by** *auto*
moreover have $\text{card } ES' < \text{card } ES$ **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

5.1.4 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
 $\exists\ ES'. (Inv\ ES' \wedge (\exists\ xrhs'. (X, xrhs') \in ES')) \wedge \text{card } ES' = 1$
(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 $?A = \text{arden-variate } X\ xrhs$
have $?P\ (\text{rexp-of-lam } ?A)$

proof –

have $L\ (\text{rexp-of-lam } ?A) = L\ (\text{lam-of } ?A)$
proof(*rule rexp-of-lam-eq-lam-set*)
show *finite* (*arden-variate* $X\ xrhs$) **using** *Inv-ES ES-single*
by (*rule-tac arden-variate-keeps-finite,*
auto simp add:Inv-def finite-rhs-def)

qed

also have $\dots = L\ ?A$

proof–

```

have lam-of ?A = ?A
proof–
  have classes-of ?A = {} using Inv-ES ES-single
    by (simp add:arden-variate-removes-cl
          self-contained-def Inv-def lefts-of-def)
  thus ?thesis
    by (auto simp only:lam-of-def classes-of-def, case-tac x, auto)
qed
thus ?thesis by simp
qed
also have ... = 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  $\exists$  (r::rexp). Lang = L r
proof -
have  $\forall$  X  $\in$  (UNIV // ( $\approx$ Lang)).  $\exists$  (reg::rexp). 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)::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

end
theory Myhill
  imports Myhill-1
begin

```

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

6.1 The scheme

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

definition

str-eq :: *string* \Rightarrow *lang* \Rightarrow *string* \Rightarrow *bool* ($- \approx -$)

where

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

The basic idea to show the finiteness of the partition induced by relation \approx_{Lang} is to attach a tag $tag(x)$ to every string x , the set of tags are carefully chosen, so that the range of tagging function tag (denoted $range(tag)$) is finite. If strings with the same tag are equivalent with respect to \approx_{Lang} , i.e. $tag(x) = tag(y) \implies x \approx_{Lang} y$ (this property is named ‘injectivity’ in the following), then it can be proved that: the partition given rise by (\approx_{Lang}) is finite.

There are two arguments for this. The first goes as the following:

1. First, the tagging function tag induces an equivalent relation $(=tag=)$ (definition of $f\text{-eq-rel}$ and lemma $equiv\text{-}f\text{-eq-rel}$).
2. It is shown that: if the range of tag is finite, the partition given rise by $(=tag=)$ is finite (lemma $finite\text{-}eq\text{-}f\text{-rel}$).
3. It is proved that if equivalent relation $R1$ is more refined than $R2$ (expressed as $R1 \subseteq R2$), and the partition induced by $R1$ is finite, then the partition induced by $R2$ is finite as well (lemma $refined\text{-}partition\text{-}finite$).
4. The injectivity assumption $tag(x) = tag(y) \implies x \approx_{Lang} y$ implies that $(=tag=)$ is more refined than (\approx_{Lang}) .
5. Combining the points above, we have: the partition induced by language $Lang$ is finite (lemma $tag\text{-}finite\text{-}imageD$).

definition

$f\text{-eq-rel} (=f=)$

where

$(=f=) = \{(x, y) \mid x \ y. \ f \ x = f \ y\}$

lemma $equiv\text{-}f\text{-eq-rel:}equiv \ UNIV \ (=f=)$

by $(auto \ simp:equiv\text{-}def \ f\text{-eq-rel}\text{-}def \ refl\text{-}on\text{-}def \ sym\text{-}def \ trans\text{-}def)$

lemma $finite\text{-}range\text{-}image: \ finite \ (range \ f) \implies \ finite \ (f \ ' \ A)$

by $(rule\text{-}tac \ B = \{y. \ \exists \ x. \ y = f \ x\} \ \mathbf{in} \ finite\text{-}subset, \ auto \ simp:image\text{-}def)$

lemma $finite\text{-}eq\text{-}f\text{-rel}:$

assumes $rng\text{-}fnt: \ finite \ (range \ tag)$

shows $finite \ (UNIV \ // \ (=tag=))$

proof –

let $?f = op \ ' \ tag$ **and** $?A = (UNIV \ // \ (=tag=))$

show $?thesis$

proof $(rule\text{-}tac \ f = ?f \ \mathbf{and} \ A = ?A \ \mathbf{in} \ finite\text{-}imageD)$

— The finiteness of f -image is a simple consequence of assumption $rng\text{-}fnt$:

show $finite \ (?f \ ' \ ?A)$

proof –

have $\forall \ X. \ ?f \ X \in (Pow \ (range \ tag))$ **by** $(auto \ simp:image\text{-}def \ Pow\text{-}def)$

moreover from $rng\text{-}fnt$ **have** $finite \ (Pow \ (range \ tag))$ **by** $simp$

ultimately have $finite \ (range \ ?f)$

by $(auto \ simp \ only:image\text{-}def \ intro:finite\text{-}subset)$

from $finite\text{-}range\text{-}image$ $[OF \ this]$ **show** $?thesis \ .$

qed

next

— The injectivity of f -image is a consequence of the definition of $(=tag=)$:

show $inj\text{-}on \ ?f \ ?A$

proof –

```

{ fix X Y
  assume X-in: X ∈ ?A
    and Y-in: Y ∈ ?A
    and tag-eq: ?f X = ?f Y
  have X = Y
  proof –
    from X-in Y-in tag-eq
    obtain x y
      where x-in: x ∈ X and y-in: y ∈ Y and eq-tg: tag x = tag y
    unfolding quotient-def Image-def str-eq-rel-def
      str-eq-def image-def f-eq-rel-def
    apply simp by blast
  with X-in Y-in show ?thesis
    by (auto simp:quotient-def str-eq-rel-def str-eq-def f-eq-rel-def)
  qed
} thus ?thesis unfolding inj-on-def by auto
qed
qed
qed

```

lemma *finite-image-finite*: $\llbracket \forall x \in A. f x \in B; \text{finite } B \rrbracket \implies \text{finite } (f \text{ ` } A)$
 by (rule finite-subset [of - B], auto)

lemma *refined-partition-finite*:
 fixes R1 R2 A
 assumes fnt: finite (A // R1)
 and refined: R1 \subseteq R2
 and eq1: equiv A R1 and eq2: equiv A R2
 shows finite (A // R2)
proof –
 let ?f = $\lambda X. \{R1 \text{ `` } \{x\} \mid x. x \in X\}$
 and ?A = (A // R2) and ?B = (A // R1)
 show ?thesis
proof(rule-tac f = ?f and A = ?A in finite-imageD)
 show finite (?f ` ?A)
proof(rule finite-subset [of - Pow ?B])
 from fnt show finite (Pow (A // R1)) by simp
 next
 from eq2
 show ?f ` A // R2 \subseteq Pow ?B
 apply (unfold image-def Pow-def quotient-def, auto)
 by (rule-tac x = xb in bexI, simp,
 unfold equiv-def sym-def refl-on-def, blast)
 qed
 next
 show inj-on ?f ?A
proof –
 { fix X Y
 assume X-in: X ∈ ?A and Y-in: Y ∈ ?A

```

    and eq-f: ?f X = ?f Y (is ?L = ?R)
  have X = Y using X-in
  proof(rule quotientE)
    fix x
    assume X = R2 “ {x} and x ∈ A with eq2
    have x-in: x ∈ X
      by (unfold equiv-def quotient-def refl-on-def, auto)
    with eq-f have R1 “ {x} ∈ ?R by auto
    then obtain y where
      y-in: y ∈ Y and eq-r: R1 “ {x} = R1 “{y} by auto
    have (x, y) ∈ R1
    proof –
      from x-in X-in y-in Y-in eq2
      have x ∈ A and y ∈ A
        by (unfold equiv-def quotient-def refl-on-def, auto)
      from eq-equiv-class-iff [OF eq1 this] and eq-r
      show ?thesis by simp
    qed
    with refined have xy-r2: (x, y) ∈ R2 by auto
    from quotient-eqI [OF eq2 X-in Y-in x-in y-in this]
    show ?thesis .
  qed
} thus ?thesis by (auto simp:inj-on-def)
qed
qed
qed

```

```

lemma equiv-lang-eq: equiv UNIV (≈Lang)
  apply (unfold equiv-def str-eq-rel-def sym-def refl-on-def trans-def)
  by blast

```

```

lemma tag-finite-imageD:
  fixes tag
  assumes rng-fnt: finite (range tag)
  — Suppose the rang of tagging fucntion tag is finite.
  and same-tag-eqvt:  $\bigwedge m n. \text{tag } m = \text{tag } (n::\text{string}) \implies m \approx \text{Lang } n$ 
  — And strings with same tag are equivalent
  shows finite (UNIV // (≈Lang))
  proof –
    let ?R1 = (=tag=)
    show ?thesis
  proof(rule-tac refined-partition-finite [of - ?R1])
    from finite-eq-f-rel [OF rng-fnt]
    show finite (UNIV // =tag=) .
  next
    from same-tag-eqvt
    show (=tag=)  $\subseteq$  (≈Lang)
    by (auto simp:f-eq-rel-def str-eq-def)
  next

```

```

from equiv-f-eq-rel
show equiv UNIV (=tag=) by blast
next
from equiv-lang-eq
show equiv UNIV ( $\approx$ Lang) by blast
qed
qed

```

A more concise, but less intelligible argument for *tag-finite-imageD* is given as the following. The basic idea is still using standard library lemma *finite-imageD*:

$$\llbracket \text{finite } (f \text{ ' } A); \text{inj-on } f \text{ } A \rrbracket \implies \text{finite } A$$

which says: if the image of injective function f over set A is finite, then A must be finite, as we did in the lemmas above.

lemma

fixes tag

assumes rng-fnt: finite (range tag)

— Suppose the range of tagging function tag is finite.

and same-tag-eqt: $\bigwedge m n. \text{tag } m = \text{tag } (n::\text{string}) \implies m \approx \text{Lang } n$

— And strings with same tag are equivalent

shows finite (UNIV // (\approx Lang))

— Then the partition generated by (\approx Lang) is finite.

proof —

— The particular f and A used in *finite-imageD* are:

let ?f = op ' tag **and** ?A = (UNIV // \approx Lang)

show ?thesis

proof (rule-tac f = ?f **and** A = ?A **in** finite-imageD)

— The finiteness of f -image is a simple consequence of assumption rng-fnt:

show finite (?f ' ?A)

proof —

have $\forall X. ?f X \in (\text{Pow } (\text{range } \text{tag}))$ **by** (auto simp:image-def Pow-def)

moreover from rng-fnt **have** finite (Pow (range tag)) **by** simp

ultimately have finite (range ?f)

by (auto simp only:image-def intro:finite-subset)

from finite-range-image [OF this] **show** ?thesis .

qed

next

— The injectivity of f is the consequence of assumption same-tag-eqt:

show inj-on ?f ?A

proof—

{ **fix** X Y

assume X-in: X \in ?A

and Y-in: Y \in ?A

and tag-eq: ?f X = ?f Y

have X = Y

proof —

from X-in Y-in tag-eq

obtain x y **where** x-in: x \in X **and** y-in: y \in Y **and** eq-tg: tag x = tag y

```

      unfolding quotient-def Image-def str-eq-rel-def str-eq-def image-def
      apply simp by blast
      from same-tag-eqvt [OF eq-tg] have  $x \approx_{Lang} y$  .
      with X-in Y-in x-in y-in
      show ?thesis by (auto simp: quotient-def str-eq-rel-def str-eq-def)
    qed
  } thus ?thesis unfolding inj-on-def by auto
qed
qed
qed

```

6.2 The proof

6.2.1 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

```

6.2.2 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)

```

6.2.3 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*)

6.2.4 The case for SEQ

definition

tag-str-SEQ $:: lang \Rightarrow lang \Rightarrow string \Rightarrow (lang \times lang \text{ set})$

where

tag-str-SEQ $L1 L2 =$

$(\lambda x. (\approx L1 \text{ “ } \{x\}, \{(\approx L2 \text{ “ } \{x - xa\}) \mid xa. xa \leq x \wedge xa \in L1\}))$

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 $\llbracket x \leq s_1; (s_1 - x) @ s_2 = y \rrbracket \implies$
 $\exists ya \leq y. (x @ ya) \in L_1 \wedge (y - ya) \in L_2$
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*:
 $tag-str-SEQ L_1 L_2 m = tag-str-SEQ L_1 L_2 n \implies m \approx(L_1 ;; L_2) n$
proof –
{ **fix** $x y z$
assume *xz-in-seq*: $x @ z \in L_1 ;; L_2$
and *tag-xy*: $tag-str-SEQ L_1 L_2 x = tag-str-SEQ L_1 L_2 y$
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 ∈ 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 [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

```

6.2.5 The case for ALT

```

definition
  tag-str-ALT :: lang ⇒ lang ⇒ string ⇒ (lang × lang)
where
  tag-str-ALT L1 L2 = (λx. (≈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 // ≈L1) × (UNIV // ≈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

```

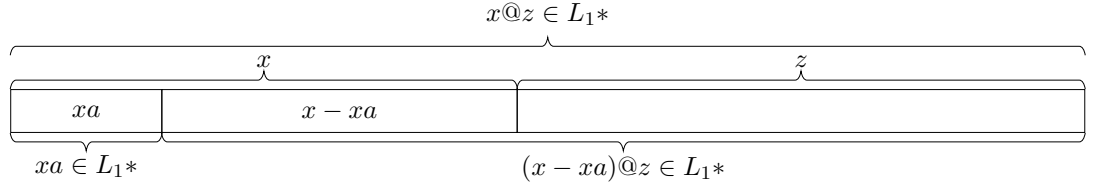
6.2.6 The case for *STAR*

This turned out to be the trickiest case. The essential goal is to prove $y @ z \in L_1^*$ under the assumptions that $x @ z \in L_1^*$ and that x and y have the same tag. The reasoning goes as the following:

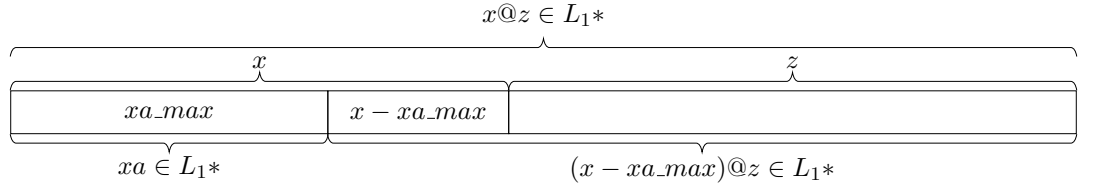
1. Since $x @ z \in L_1^*$ holds, a prefix xa of x can be found such that $xa \in L_1^*$ and $(x - xa)@z \in L_1^*$, as shown in Fig. 1(a)(a). Such a prefix always exists, $xa = []$, for example, is one.
2. There could be many but finite many of such xa , from which we can find the longest and name it $xa-max$, as shown in Fig. 1(b)(b).
3. The next step is to split z into za and zb such that $(x - xa-max) @ za \in L_1$ and $zb \in L_1^*$ as shown in Fig. 1(d)(d). Such a split always exists because:
 - (a) Because $(x - xa-max) @ z \in L_1^*$, it can always be split into prefix a and suffix b , such that $a \in L_1$ and $b \in L_1^*$, as shown in Fig. 1(c)(c).
 - (b) But the prefix a CANNOT be shorter than $x - xa-max$, otherwise $xa-max$ is not the max in its kind.
 - (c) Now, za is just $a - (x - xa-max)$ and zb is just b .
4. By the assumption that x and y have the same tag, the structure on $x @ z$ can be transferred to $y @ z$ as shown in Fig. 1(e)(e). The detailed steps are:
 - (a) A y -prefix ya corresponding to xa can be found, which satisfies conditions: $ya \in L_1^*$ and $(y - ya)@za \in L_1$.
 - (b) Since we already know $zb \in L_1^*$, we get $(y - ya)@za@zb \in L_1^*$, and this is just $(y - ya)@z \in L_1^*$.
 - (c) With fact $ya \in L_1^*$, we finally get $y@z \in L_1^*$.

The formal proof of lemma *tag-str-STAR-injI* faithfully follows this informal argument while the tagging function *tag-str-STAR* is defined to make the transfer in step 4 feasible.

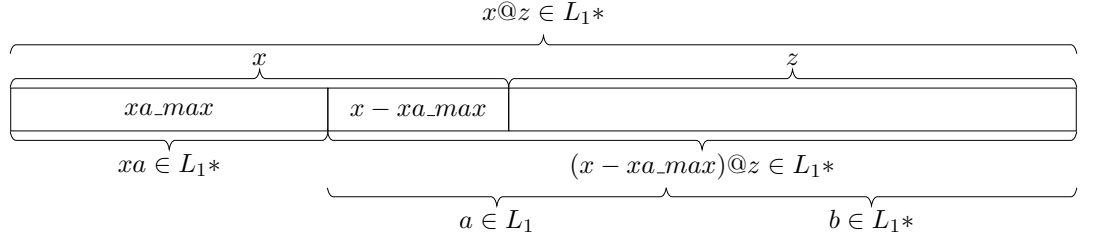
definition



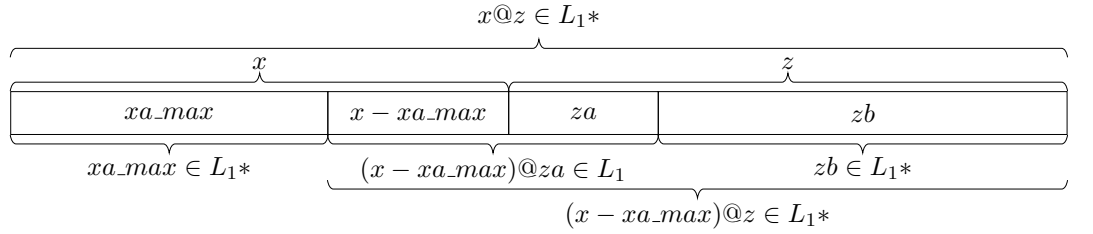
(a) First split



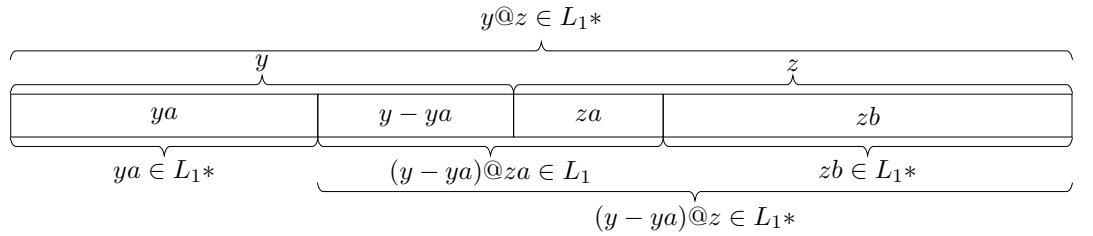
(b) Max split



(c) Max split with a and b



(d) Last split



(e) Transferring to y

Figure 1: The case for *STAR*

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

A technical lemma.

lemma *finite-set-has-max*: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \Longrightarrow$
 $(\exists \text{ max} \in A. \forall a \in A. f a \leq (f \text{ max} :: \text{nat}))$
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** *bestI*, *auto*)
 next
 case *False*
 with *prems* **obtain** *max*
 where *h1*: *max* \in A
 and *h2*: $\forall a \in A. f a \leq f \text{ max}$ **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** *bestI*, *auto*)
 next
 assume \neg (f a \leq f *max*)
 thus ?*thesis* **using** *h2* **by** (*rule-tac* x = a **in** *bestI*, *auto*)
 qed
qed
qed

Technical lemma.

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*:
fixes v w
assumes *eq-tag*: *tag-str-STAR* L1 v = *tag-str-STAR* L1 w
shows (v::string) $\approx_{(L1\star)}$ w

proof –
 According to the definition of \approx_{Lang} , proving v $\approx_{(L1\star)}$ w amounts to
 — showing: for any string u, if v @ u \in (L1 \star) then w @ u \in (L1 \star) and vice
 versa. The reasoning pattern for both directions are the same, as derived
 in the following:

{ **fix** x y z
 assume *xz-in-star*: x @ z \in L1 \star
 and *tag-xy*: *tag-str-STAR* L1 x = *tag-str-STAR* L1 y

have $y @ z \in L_1^*$
proof(*cases* $x = []$)
— The degenerated case when x is a null string is easy to prove:
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
— The case when x is not null, and $x @ z$ is in L_1^* ,

case *False*
Since $x @ z \in L_1^*$, x can always be split by a prefix xa together with its
suffix $x - xa$, such that both xa and $(x - xa) @ z$ are in L_1^* , and there
could be many such splittings. Therefore, the following set $?S$ is nonempty,
and finite as well:
let $?S = \{xa. xa < x \wedge xa \in L_1^* \wedge (x - xa) @ z \in L_1^*\}$
have *finite* $?S$
by (*rule-tac* $B = \{xa. xa < x\}$ **in** *finite-subset*,
auto simp:finite-strict-prefix-set)
moreover **have** $?S \neq \{\}$ **using** *False xz-in-star*
by (*simp*, *rule-tac* $x = []$ **in** *exI*, *auto simp:strict-prefix-def*)
— Since $?S$ is finite, we can always single out the longest and name it $xa-max$:

ultimately **have** $\exists xa-max \in ?S. \forall xa \in ?S. length\ xa \leq length\ xa-max$
using *finite-set-has-max* **by** *blast*
then **obtain** $xa-max$
where $h1: xa-max < x$
and $h2: xa-max \in L_1^*$
and $h3: (x - xa-max) @ z \in L_1^*$
and $h4: \forall xa < x. xa \in L_1^* \wedge (x - xa) @ z \in L_1^*$
 $\rightarrow length\ xa \leq length\ xa-max$

by *blast*
— By the equality of tags, the counterpart of $xa-max$ among y -prefixes, named
 ya , can be found:
obtain ya
where $h5: ya < y$ **and** $h6: ya \in L_1^*$
and $eq-xya: (x - xa-max) \approx_{L_1} (y - ya)$
proof—
from *tag-xy* **have** $\{\approx_{L_1} \{x - xa\} \mid xa. xa < x \wedge xa \in L_1^*\} =$
 $\{\approx_{L_1} \{y - xa\} \mid xa. xa < y \wedge xa \in L_1^*\}$ (**is** *?left = ?right*)
by (*auto simp:tag-str-STAR-def*)
moreover **have** $\approx_{L_1} \{x - xa-max\} \in ?left$ **using** $h1\ h2$ **by** *auto*
ultimately **have** $\approx_{L_1} \{x - xa-max\} \in ?right$ **by** *simp*
with *prems* **show** *?thesis* **apply**
(*simp add:Image-def str-eq-rel-def str-eq-def*) **by** *blast*
qed
— If the following proposition can be proved, then the *?thesis: y @ z \in L_1^**
is just a simple consequence.
have $(y - ya) @ z \in L_1^*$
proof—
— The idea is to split the suffix z into za and zb , such that:

obtain $za\ zb$ **where** $eq\text{-}zab: z = za @ zb$
and $l\text{-}za: (y - ya)@za \in L_1$ **and** $ls\text{-}zb: zb \in L_1\star$
proof –
– Since $(x - xa\text{-}max) @ z$ is in $L_1\star$, it can be split into a and b such that:

from $h1$ **have** $(x - xa\text{-}max) @ z \neq []$
by $(auto\ simp:strict\text{-}prefix\text{-}def\ elim:prefixE)$
from $star\text{-}decom$ [$OF\ h3\ this$]
obtain $a\ b$ **where** $a\text{-}in: a \in L_1$
and $a\text{-}neg: a \neq []$ **and** $b\text{-}in: b \in L_1\star$
and $ab\text{-}max: (x - xa\text{-}max) @ z = a @ b$ **by** $blast$
– Now the candidates for za and zb are found:

let $?za = a - (x - xa\text{-}max)$ **and** $?zb = b$
have $pfx: (x - xa\text{-}max) \leq a$ (**is** $?P1$)
and $eq\text{-}z: z = ?za @ ?zb$ (**is** $?P2$)
proof –
– Since $(x - xa\text{-}max) @ z = a @ b$, the string $(x - xa\text{-}max) @ z$ could be splitted in two ways:

have $((x - xa\text{-}max) \leq a \wedge (a - (x - xa\text{-}max)) @ b = z) \vee$
 $(a < (x - xa\text{-}max) \wedge ((x - xa\text{-}max) - a) @ z = b)$
using $app\text{-}eq\text{-}dest$ [$OF\ ab\text{-}max$] **by** $(auto\ simp:strict\text{-}prefix\text{-}def)$
moreover {
– However, the undesired way can be refuted by absurdity:

assume $np: a < (x - xa\text{-}max)$
and $b\text{-}eqs: ((x - xa\text{-}max) - a) @ z = b$
have $False$
proof –
let $?xa\text{-}max' = xa\text{-}max @ a$
have $?xa\text{-}max' < x$
using $np\ h1$ **by** $(clarsimp\ simp:strict\text{-}prefix\text{-}def\ diff\text{-}prefix)$
moreover **have** $?xa\text{-}max' \in L_1\star$
using $a\text{-}in\ h2$ **by** $(simp\ add:star\text{-}intro3)$
moreover **have** $(x - ?xa\text{-}max') @ z \in L_1\star$
using $b\text{-}eqs\ b\text{-}in\ np\ h1$ **by** $(simp\ add:diff\text{-}diff\text{-}appd)$
moreover **have** $\neg (length\ ?xa\text{-}max' \leq length\ xa\text{-}max)$
using $a\text{-}neg$ **by** $simp$
ultimately **show** $?thesis$ **using** $h4$ **by** $blast$
qed }
– Now it can be shown that the splitting goes the way we desired.

ultimately **show** $?P1$ **and** $?P2$ **by** $auto$
qed
hence $(x - xa\text{-}max)@?za \in L_1$ **using** $a\text{-}in$ **by** $(auto\ elim:prefixE)$
– Now candidates $?za$ and $?zb$ have all the required properties.

with $eq\text{-}xya$ **have** $(y - ya) @ ?za \in L_1$
by $(auto\ simp:str\text{-}eq\text{-}def\ str\text{-}eq\text{-}rel\text{-}def)$
with $eq\text{-}z$ **and** $b\text{-}in$ **prems**
show $?thesis$ **by** $blast$
qed
– From the properties of za and zb such obtained, $?thesis$ can be shown easily.

```

    from step [OF l-za ls-zb]
    have ((y - ya) @ za) @ zb ∈ L1★ .
    with eq-zab show ?thesis by simp
qed
with h5 h6 show ?thesis
  by (drule-tac star-intro1, auto simp:strict-prefix-def elim:prefixE)
qed
}
— By instantiating the reasoning pattern just derived for both directions:
from this [OF - eq-tag] and this [OF - eq-tag [THEN sym]]
— The thesis is proved as a trival consequence:
show ?thesis by (unfold str-eq-def str-eq-rel-def, blast)
qed

```

lemma — The oringal version with a poor readability
fixes $v w$
assumes $eq\text{-tag}: tag\text{-str-STAR } L_1 v = tag\text{-str-STAR } L_1 w$
shows $(v::string) \approx(L_1★) w$

proof—
 According to the definition of $\approx Lang$, proving $v \approx(L_1★) w$ amounts to
 — showing: for any string u , if $v @ u \in (L_1★)$ then $w @ u \in (L_1★)$ and vice
 versa. The reasoning pattern for both directions are the same, as derived
 in the following:

```

{ fix x y z
  assume xz-in-star: x @ z ∈ L1★
  and tag-xy: tag-str-STAR L1 x = tag-str-STAR L1 y
  have y @ z ∈ L1★
  proof(cases x = [])
  — The degenerated case when  $x$  is a null string is easy to prove:
  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
 — The case when x is not null, and $x @ z$ is in $L_1★$,

```

case False
obtain x-max
  where h1: x-max < x
  and h2: x-max ∈ L1★
  and h3: (x - x-max) @ z ∈ L1★
  and h4:∀ xa < x. xa ∈ L1★ ∧ (x - xa) @ z ∈ L1★
  → length xa ≤ length x-max

```

proof—
let $?S = \{xa. xa < x \wedge xa \in L_1★ \wedge (x - xa) @ z \in L_1★\}$
have *finite* $?S$
by (rule-tac $B = \{xa. xa < x\}$ **in** *finite-subset*,

auto simp:finite-strict-prefix-set)

moreover have $?S \neq \{\}$ using *False xz-in-star*
 by (*simp, rule-tac x = [] in exI, auto simp:strict-prefix-def*)
 ultimately have $\exists \text{max} \in ?S. \forall a \in ?S. \text{length } a \leq \text{length } \text{max}$
 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 $\{\approx_{L_1} \text{ “ } \{x - xa\} \mid xa. xa < x \wedge xa \in L_1^\star \} =$
 $\{\approx_{L_1} \text{ “ } \{y - xa\} \mid xa. xa < y \wedge xa \in L_1^\star \}$ (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 $?x\text{-max}' = x\text{-max} @ a$
 have $?x\text{-max}' < x$
 using *np h1* by (*clarsimp simp:strict-prefix-def diff-prefix*)
 moreover have $?x\text{-max}' \in L_1^\star$
 using *a-in h2* by (*simp add:star-intro3*)
 moreover have $(x - ?x\text{-max}') @ z \in L_1^\star$
 using *b-eqs b-in np h1* by (*simp add:diff-diff-appd*)
 moreover have $\neg (\text{length } ?x\text{-max}' \leq \text{length } x\text{-max})$
 using *a-neq* by *simp*
 ultimately show *?thesis* using *h4* by *blast*
 qed
 } ultimately show *?thesis* by *blast*

qed

then obtain *za* where *z-decom*: $z = za @ b$
 and *x-za*: $(x - x\text{-max}) @ za \in L_1$
 using *a-in* by (*auto elim:prefixE*)


```

    from x-za h7 have (y - ya) @ za ∈ 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
}
— By instantiating the reasoning pattern just derived for both directions:
from this [OF - eq-tag] and this [OF - eq-tag [THEN sym]]
— The thesis is proved as a trival consequence:
show ?thesis by (unfold str-eq-def str-eq-rel-def, blast)
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

```

6.2.7 The conclusion

```

lemma rexp-imp-finite:
  fixes r::rexp
  shows finite (UNIV // ≈(L r))
by (induct r) (auto)

end

```