

The Myhill-Nerode Theorem Based on Regular Expressions

Chunhan Wu, Xingyuan Zhang and Christian Urban

September 13, 2023

Abstract

There are many proofs of the Myhill-Nerode theorem using automata. In this library we give a proof entirely based on regular expressions, since regularity of languages can be conveniently defined using regular expressions (it is more painful in HOL to define regularity in terms of automata). We prove the first direction of the Myhill-Nerode theorem by solving equational systems that involve regular expressions. For the second direction we give two proofs: one using tagging-functions and another using partial derivatives. We also establish various closure properties of regular languages.¹

Contents

1	“Summation” for regular expressions	2
2	First direction of MN: <i>finite partition</i> \Rightarrow <i>regular language</i>	3
2.1	Equational systems	4
2.2	Arden Operation on equations	5
2.3	Substitution Operation on equations	5
2.4	While-combinator and invariants	5
2.5	Initial Equational Systems	8
2.6	Iterations	10
2.7	The conclusion of the first direction	16
3	Second direction of MN: <i>regular language</i> \Rightarrow <i>finite partition</i>	18
3.1	Tagging functions	18
3.2	Base cases: <i>Zero</i> , <i>One</i> and <i>Atom</i>	21
3.3	Case for <i>Plus</i>	22
3.4	Case for <i>Times</i>	22
3.5	Case for <i>Star</i>	25
3.6	The conclusion of the second direction	28

¹Most details of the theories are described in the paper [2].

4	The theorem	28
4.1	Second direction proved using partial derivatives	28
5	Closure properties of regular languages	29
5.1	Closure under \cup , \cdot and \star	29
5.2	Closure under complementation	30
5.3	Closure under $-$ and \cap	30
5.4	Closure under string reversal	31
5.5	Closure under left-quotients	32
5.6	Finite and co-finite sets are regular	32
5.7	Continuation lemma for showing non-regularity of languages .	33
5.8	The language $a^n b^n$ is not regular	34
6	Closure under <i>SUBSEQ</i> and <i>SUPSEQ</i>	35
6.1	Sub- and Supersequences	36
6.2	Regular expression that recognises every character	37
6.3	Closure of <i>SUBSEQ</i> and <i>SUPSEQ</i>	39
7	Tools for showing non-regularity of a language	40
7.1	Auxiliary material	40
7.2	Non-regularity by giving an infinite set of equivalence classes	41
7.3	The Pumping Lemma	42
7.4	Examples	44

```

theory Folds
imports Regular-Sets.Regular-Exp
begin

```

1 “Summation” for regular expressions

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

definition

folds :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a set \Rightarrow 'b

where

folds f z S \equiv *SOME x. fold-graph f z S x*

Plus-combination for a set of regular expressions

abbreviation

Setalt :: 'a rexp set \Rightarrow 'a rexp (\uplus - [1000] 999)

where

$\uplus A$ \equiv *folds Plus Zero A*

For finite sets, *Setalt* is preserved under *lang*.

```

lemma folds-plus-simp [simp]:
  fixes rs::('a rexp) set
  assumes a: finite rs
  shows lang ( $\bigoplus$  rs) =  $\bigcup$  (lang ' rs)
unfolding folds-def
apply(rule set-eqI)
apply(rule someI2-ex)
apply(rule-tac finite-imp-fold-graph[OF a])
apply(erule fold-graph.induct)
apply(auto)
done

end

```

```

theory Myhill-1
imports Folds
         HOL-Library.While-Combinator
begin

```

2 First direction of MN: *finite partition* \Rightarrow *regular language*

```

notation
  conc (infixr · 100) and
  star (-★ [101] 102)

```

```

lemma Pair-Collect [simp]:
  shows  $(x, y) \in \{(x, y). P\ x\ y\} \longleftrightarrow P\ x\ y$ 
by simp

```

Myhill-Nerode relation

```

definition
  str-eq :: 'a lang  $\Rightarrow$  ('a list  $\times$  'a list) set ( $\approx$ - [100] 100)
where
   $\approx A \equiv \{(x, y). (\forall z. x @ z \in A \longleftrightarrow y @ z \in A)\}$ 

```

```

abbreviation
  str-eq-applied :: 'a list  $\Rightarrow$  'a lang  $\Rightarrow$  'a list  $\Rightarrow$  bool (-  $\approx$ - -)
where
   $x \approx A\ y \equiv (x, y) \in \approx A$ 

```

```

lemma str-eq-conv-Derivs:
  str-eq A =  $\{(u, v). \text{Derivs } u\ A = \text{Derivs } v\ A\}$ 
by (auto simp: str-eq-def Derivs-def)

```

```

definition
  finals :: 'a lang  $\Rightarrow$  'a lang set

```

where

$finals\ A \equiv \{\approx A\ \{\{s\} \mid s . s \in A\}$

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

shows $A = \bigcup (finals\ A)$

unfolding *finals-def*

unfolding *Image-def*

unfolding *str-eq-def*

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

lemma *finals-in-partitions:*

shows $finals\ A \subseteq (UNIV // \approx A)$

unfolding *finals-def quotient-def*

by *auto*

2.1 Equational systems

The two kinds of terms in the rhs of equations.

datatype *'a trm =*

Lam 'a rexp

| *Trn 'a lang 'a rexp*

fun

lang-trm::'a trm \Rightarrow 'a lang

where

lang-trm (Lam r) = lang r

| *lang-trm (Trn X r) = X \cdot lang r*

fun

lang-rhs::('a trm) set \Rightarrow 'a lang

where

lang-rhs rhs = \bigcup (lang-trm ` rhs)

lemma *lang-rhs-set:*

shows $lang-rhs\ \{Trn\ X\ r \mid r. P\ r\} = \bigcup \{lang-trm\ (Trn\ X\ r) \mid r. P\ r\}$

by (*auto*)

lemma *lang-rhs-union-distrib:*

shows $lang-rhs\ A \cup lang-rhs\ B = lang-rhs\ (A \cup B)$

by *simp*

Transitions between equivalence classes

definition

transition :: 'a lang \Rightarrow 'a \Rightarrow 'a lang \Rightarrow bool (- \models ->- [100,100,100] 100)

where

Y $\models c \Rightarrow X \equiv Y \cdot \{[c]\} \subseteq X$

Initial equational system

definition

Init-rhs CS X \equiv
 if ($\square \in X$) then
 $\{Lam\ One\} \cup \{Trn\ Y\ (Atom\ c) \mid Y\ c.\ Y \in CS \wedge Y \models c \Rightarrow X\}$
 else
 $\{Trn\ Y\ (Atom\ c) \mid Y\ c.\ Y \in CS \wedge Y \models c \Rightarrow X\}$

definition

Init CS $\equiv \{(X, Init\text{-}rhs\ CS\ X) \mid X.\ X \in CS\}$

2.2 Arden Operation on equations

fun

Append-rexp $:: 'a\ rexp \Rightarrow 'a\ trm \Rightarrow 'a\ trm$

where

Append-rexp r (Lam rexp) = *Lam (Times rexp r)*
| Append-rexp r (Trn X rexp) = *Trn X (Times rexp r)*

definition

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

definition

Arden X rhs \equiv
Append-rexp-rhs (rhs - {Trn X r | r. Trn X r \in rhs}) (Star (\bigoplus {r. Trn X r \in rhs}))

2.3 Substitution Operation on equations

definition

Subst rhs X xrhs \equiv
 $(rhs - \{Trn\ X\ r \mid r.\ Trn\ X\ r \in rhs\}) \cup (Append\text{-}rexp\text{-}rhs\ xrhs\ (\bigoplus\ \{r.\ Trn\ X\ r \in rhs\}))$

definition

Subst-all $:: ('a\ lang \times ('a\ trm)\ set)\ set \Rightarrow 'a\ lang \Rightarrow ('a\ trm)\ set \Rightarrow ('a\ lang \times ('a\ trm)\ set)\ set$

where

Subst-all ES X xrhs $\equiv \{(Y, Subst\ yrhs\ X\ xrhs) \mid Y\ yrhs.\ (Y, yrhs) \in ES\}$

definition

Remove ES X xrhs \equiv
Subst-all (ES - {(X, xrhs)}) X (Arden X xrhs)

2.4 While-combinator and invariants

definition

Iter X ES $\equiv (let\ (Y, yrhs) = SOME\ (Y, yrhs).\ (Y, yrhs) \in ES \wedge X \neq Y$
in Remove ES Y yrhs)

lemma *IterI2*:

assumes $(Y, yrhs) \in ES$
and $X \neq Y$
and $\bigwedge Y yrhs. \llbracket (Y, yrhs) \in ES; X \neq Y \rrbracket \implies Q$ (*Remove ES Y yrhs*)
shows Q (*Iter X ES*)
unfolding *Iter-def* **using** *assms*
by (*rule-tac a=(Y, yrhs) in someI2*) (*auto*)

abbreviation

Cond ES \equiv *card ES* $\neq 1$

definition

Solve X ES \equiv *while Cond (Iter X) ES*

definition

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

definition

soundness ES $\equiv \forall (X, rhs) \in ES. X = \text{lang-rhs } rhs$

definition

ardenable rhs $\equiv (\forall Y r. \text{Trn } Y r \in rhs \longrightarrow [] \notin \text{lang } r)$

definition

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

definition

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

lemma *finite-rhs-def2*:

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

unfolding *finite-rhs-def* **by** *auto*

definition

rhss rhs $\equiv \{X \mid \exists r. \text{Trn } X r \in rhs\}$

definition

lhss ES $\equiv \{Y \mid \exists yrhs. (Y, yrhs) \in ES\}$

definition

validity ES $\equiv \forall (X, rhs) \in ES. \text{rhss } rhs \subseteq \text{lhss } ES$

lemma *rhss-union-distrib*:

shows $\text{rhss } (A \cup B) = \text{rhss } A \cup \text{rhss } B$

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

lemma *lhss-union-distrib*:

shows $\text{lhss } (A \cup B) = \text{lhss } A \cup \text{lhss } B$

by (auto simp add: lhss-def)

definition

invariant ES \equiv *finite ES*
 \wedge *finite-rhs ES*
 \wedge *soundness ES*
 \wedge *distinctness ES*
 \wedge *ardenable-all ES*
 \wedge *validity ES*

lemma invariantI:

assumes *soundness ES finite ES distinctness ES ardenable-all ES*
finite-rhs ES validity ES
shows *invariant ES*
using *assms* by (simp add: *invariant-def*)

declare [[*simplproc add: finite-Collect*]]

lemma finite-Trn:

assumes *fin: finite rhs*
shows *finite {r. Trn Y r \in rhs}*
using *assms* by (auto intro!: *finite-vimageI simp add: inj-on-def*)

lemma finite-Lam:

assumes *fin: finite rhs*
shows *finite {r. Lam r \in rhs}*
using *assms* by (auto intro!: *finite-vimageI simp add: inj-on-def*)

lemma trm-soundness:

assumes *finite:finite rhs*
shows *lang-rhs* ($\{Trn X r \mid r. Trn X r \in rhs\}$) = $X \cdot (lang (\bigoplus \{r. Trn X r \in rhs\}))$
proof –
have *finite {r. Trn X r \in rhs}*
by (rule *finite-Trn[OF finite]*)
then show *lang-rhs* ($\{Trn X r \mid r. Trn X r \in rhs\}$) = $X \cdot (lang (\bigoplus \{r. Trn X r \in rhs\}))$
by (*simp only: lang-rhs-set lang-trm.simps*) (auto simp add: *conc-def*)
qed

lemma lang-of-append-rexp:

lang-trm (*Append-rexp r trm*) = *lang-trm trm* \cdot *lang r*
by (*induct rule: Append-rexp.induct*)
(auto simp add: *conc-assoc*)

lemma lang-of-append-rexp-rhs:

$lang-rhs (Append-rexp-rhs\ rhs\ r) = lang-rhs\ rhs \cdot lang\ r$
unfolding *Append-rexp-rhs-def*
by (*auto simp add: conc-def lang-of-append-rexp*)

2.5 Intial Equational Systems

lemma *defined-by-str:*
assumes $s \in X\ X \in UNIV // \approx A$
shows $X = \approx A \text{ “ } \{s\}$
using *assms*
unfolding *quotient-def Image-def str-eq-def*
by *auto*

lemma *every-eqclass-has-transition:*
assumes *has-str:* $s @ [c] \in X$
and *in-CS:* $X \in UNIV // \approx A$
obtains Y **where** $Y \in UNIV // \approx A$ **and** $Y \cdot \{[c]\} \subseteq X$ **and** $s \in Y$
proof –
define Y **where** $Y = \approx A \text{ “ } \{s\}$
have $Y \in UNIV // \approx A$
unfolding *Y-def quotient-def* **by** *auto*
moreover
have $X = \approx A \text{ “ } \{s @ [c]\}$
using *has-str in-CS defined-by-str* **by** *blast*
then have $Y \cdot \{[c]\} \subseteq X$
unfolding *Y-def Image-def conc-def*
unfolding *str-eq-def*
by *clarsimp*
moreover
have $s \in Y$ **unfolding** *Y-def*
unfolding *Image-def str-eq-def* **by** *simp*
ultimately show thesis using that by blast
qed

lemma *l-eq-r-in-eqs:*
assumes *X-in-eqs:* $(X, rhs) \in Init (UNIV // \approx A)$
shows $X = lang-rhs\ rhs$
proof
show $X \subseteq lang-rhs\ rhs$
proof
fix x
assume *in-X:* $x \in X$
{ assume *empty:* $x = []$
then have $x \in lang-rhs\ rhs$ **using** *X-in-eqs in-X*
unfolding *Init-def Init-rhs-def*
by *auto*
}
moreover
{ assume *not-empty:* $x \neq []$

```

then obtain  $s\ c$  where  $decom: x = s @ [c]$ 
  using  $rev-cases$  by  $blast$ 
have  $X \in UNIV // \approx A$  using  $X-in-eqs$  unfolding  $Init-def$  by  $auto$ 
then obtain  $Y$  where  $Y \in UNIV // \approx A$   $Y \cdot \{[c]\} \subseteq X$   $s \in Y$ 
  using  $decom\ in-X\ every-eqclass-has-transition$  by  $metis$ 
then have  $x \in lang-rhs \{Trn\ Y\ (Atom\ c) \mid Y\ c.\ Y \in UNIV // \approx A \wedge Y \models c \Rightarrow$ 
 $X\}$ 
  unfolding  $transition-def$ 
  using  $decom$  by  $(force\ simp\ add: conc-def)$ 
then have  $x \in lang-rhs\ rhs$  using  $X-in-eqs\ in-X$ 
  unfolding  $Init-def\ Init-rhs-def$  by  $simp$ 
}
ultimately show  $x \in lang-rhs\ rhs$  by  $blast$ 
qed
next
show  $lang-rhs\ rhs \subseteq X$  using  $X-in-eqs$ 
  unfolding  $Init-def\ Init-rhs-def\ transition-def$ 
  by  $auto$ 
qed

```

```

lemma  $finite-Init-rhs:$ 
  fixes  $CS::('a::finite)\ lang$   $set$ 
  assumes  $finite: finite\ CS$ 
  shows  $finite\ (Init-rhs\ CS\ X)$ 
using  $assms$  unfolding  $Init-rhs-def\ transition-def$  by  $simp$ 

```

```

lemma  $Init-ES-satisfies-invariant:$ 
  fixes  $A::('a::finite)\ lang$ 
  assumes  $finite-CS: finite\ (UNIV // \approx A)$ 
  shows  $invariant\ (Init\ (UNIV // \approx A))$ 
proof  $(rule\ invariantI)$ 
  show  $soundness\ (Init\ (UNIV // \approx A))$ 
    unfolding  $soundness-def$ 
    using  $l-eq-r-in-eqs$  by  $auto$ 
  show  $finite\ (Init\ (UNIV // \approx A))$  using  $finite-CS$ 
    unfolding  $Init-def$  by  $simp$ 
  show  $distinctness\ (Init\ (UNIV // \approx A))$ 
    unfolding  $distinctness-def\ Init-def$  by  $simp$ 
  show  $ardenable-all\ (Init\ (UNIV // \approx A))$ 
    unfolding  $ardenable-all-def\ Init-def\ Init-rhs-def\ ardenable-def$ 
    by  $auto$ 
  show  $finite-rhs\ (Init\ (UNIV // \approx A))$ 
    using  $finite-Init-rhs[OF\ finite-CS]$ 
    unfolding  $finite-rhs-def\ Init-def$  by  $auto$ 
  show  $validity\ (Init\ (UNIV // \approx A))$ 
    unfolding  $validity-def\ Init-def\ Init-rhs-def\ rhss-def\ lhss-def$ 
    by  $auto$ 

```

qed

2.6 Iterations

lemma *Arden-preserves-soundness*:

assumes *l-eq-r*: $X = \text{lang-rhs } rhs$

and *not-empty*: *ardenable rhs*

and *finite*: *finite rhs*

shows $X = \text{lang-rhs } (\text{Arden } X \text{ } rhs)$

proof –

define *A* **where** $A = \text{lang } (\biguplus \{r. \text{Trn } X \text{ } r \in rhs\})$

define *b* **where** $b = \{\text{Trn } X \text{ } r \mid r. \text{Trn } X \text{ } r \in rhs\}$

define *B* **where** $B = \text{lang-rhs } (rhs - b)$

have *not-empty2*: $\square \notin A$

using *finite-Trn*[*OF finite*] *not-empty*

unfolding *A-def* *ardenable-def* **by** *simp*

have $X = \text{lang-rhs } rhs$ **using** *l-eq-r* **by** *simp*

also have $\dots = \text{lang-rhs } (b \cup (rhs - b))$ **unfolding** *b-def* **by** *auto*

also have $\dots = \text{lang-rhs } b \cup B$ **unfolding** *B-def* **by** (*simp only: lang-rhs-union-distrib*)

also have $\dots = X \cdot A \cup B$

unfolding *b-def*

unfolding *trm-soundness*[*OF finite*]

unfolding *A-def*

by *blast*

finally have $X = X \cdot A \cup B$.

then have $X = B \cdot A^*$

by (*simp add: reversed-Arden*[*OF not-empty2*])

also have $\dots = \text{lang-rhs } (\text{Arden } X \text{ } rhs)$

unfolding *Arden-def* *A-def* *B-def* *b-def*

by (*simp only: lang-of-append-rexp-rhs lang.simps*)

finally show $X = \text{lang-rhs } (\text{Arden } X \text{ } rhs)$ **by** *simp*

qed

lemma *Append-preserves-finite*:

finite rhs \implies *finite* (*Append-rexp-rhs rhs r*)

by (*auto simp: Append-rexp-rhs-def*)

lemma *Arden-preserves-finite*:

finite rhs \implies *finite* (*Arden X rhs*)

by (*auto simp: Arden-def Append-preserves-finite*)

lemma *Append-preserves-ardenable*:

ardenable rhs \implies *ardenable* (*Append-rexp-rhs rhs r*)

apply (*auto simp: ardenable-def Append-rexp-rhs-def*)

by (*case-tac x, auto simp: conc-def*)

lemma *ardenable-set-sub*:

ardenable rhs \implies *ardenable* (*rhs - A*)

by (*auto simp: ardenable-def*)

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

lemma *Arden-preserves-ardenable*:
 $\text{ardenable } rhs \implies \text{ardenable } (\text{Arden } X \text{ } rhs)$
by (*simp only:Arden-def Append-preserves-ardenable ardenable-set-sub*)

lemma *Subst-preserves-ardenable*:
 $\llbracket \text{ardenable } rhs; \text{ardenable } xrhs \rrbracket \implies \text{ardenable } (\text{Subst } rhs \text{ } X \text{ } xrhs)$
by (*simp only: Subst-def Append-preserves-ardenable ardenable-set-union ardenable-set-sub*)

lemma *Subst-preserves-soundness*:
assumes *substor*: $X = \text{lang-rhs } xrhs$
and *finite*: *finite* *rhs*
shows $\text{lang-rhs } (\text{Subst } rhs \text{ } X \text{ } xrhs) = \text{lang-rhs } rhs$ (**is** $?Left = ?Right$)
proof –
define *A* **where** $A = \text{lang-rhs } (rhs - \{Trn \ X \ r \mid r. Trn \ X \ r \in rhs\})$
have $?Left = A \cup \text{lang-rhs } (\text{Append-rexp-rhs } xrhs \ (\biguplus \{r. Trn \ X \ r \in rhs\}))$
unfolding *Subst-def*
unfolding *lang-rhs-union-distrib[symmetric]*
by (*simp add: A-def*)
moreover have $?Right = A \cup \text{lang-rhs } \{Trn \ X \ r \mid r. Trn \ X \ r \in rhs\}$
proof –
have $rhs = (rhs - \{Trn \ X \ r \mid r. Trn \ X \ r \in rhs\}) \cup (\{Trn \ X \ r \mid r. Trn \ X \ r \in rhs\})$ **by** *auto*
thus *?thesis*
unfolding *A-def*
unfolding *lang-rhs-union-distrib*
by *simp*
qed
moreover
have $\text{lang-rhs } (\text{Append-rexp-rhs } xrhs \ (\biguplus \{r. Trn \ X \ r \in rhs\})) = \text{lang-rhs } \{Trn \ X \ r \mid r. Trn \ X \ r \in rhs\}$
using *finite substor* **by** (*simp only: lang-of-append-rexp-rhs trm-soundness*)
ultimately show *?thesis* **by** *simp*
qed

lemma *Subst-preserves-finite-rhs*:
 $\llbracket \text{finite } rhs; \text{finite } yrhs \rrbracket \implies \text{finite } (\text{Subst } rhs \text{ } Y \text{ } yrhs)$
by (*auto simp: Subst-def Append-preserves-finite*)

lemma *Subst-all-preserves-finite*:
assumes *finite*: *finite* *ES*
shows *finite* $(\text{Subst-all } ES \text{ } Y \text{ } yrhs)$
using *assms* **unfolding** *Subst-all-def* **by** *simp*

declare $[[\text{simproc del: finite-Collect}]]$

lemma *Subst-all-preserves-finite-rhs:*

$[[\text{finite-rhs ES}; \text{finite yrhs}]] \implies \text{finite-rhs (Subst-all ES Y yrhs)}$
by (auto intro:Subst-preserves-finite-rhs simp add:Subst-all-def finite-rhs-def)

lemma *append-rhs-preserves-cls:*

$\text{rhss (Append-rexp-rhs rhs r)} = \text{rhss rhs}$
apply (auto simp: rhss-def Append-rexp-rhs-def)
apply (case-tac xa, auto simp: image-def)
by (rule-tac x = Times ra r in exI, rule-tac x = Trn x ra in bexI, simp+)

lemma *Arden-removes-cl:*

$\text{rhss (Arden Y yrhs)} = \text{rhss yrhs} - \{Y\}$
apply (simp add:Arden-def append-rhs-preserves-cls)
by (auto simp: rhss-def)

lemma *lhss-preserves-cls:*

$\text{lhss (Subst-all ES Y yrhs)} = \text{lhss ES}$
by (auto simp: lhss-def Subst-all-def)

lemma *Subst-updates-cls:*

$X \notin \text{rhss xrhs} \implies$
 $\text{rhss (Subst rhs X xrhs)} = \text{rhss rhs} \cup \text{rhss xrhs} - \{X\}$
apply (simp only:Subst-def append-rhs-preserves-cls rhss-union-distrib)
by (auto simp: rhss-def)

lemma *Subst-all-preserves-validity:*

assumes sc: validity (ES \cup $\{(Y, \text{yrhs})\}$) (is validity ?A)
shows validity (Subst-all ES Y (Arden Y yrhs)) (is validity ?B)
proof –
 { **fix** X xrhs'
 assume (X, xrhs') \in ?B
 then obtain rhss
 where xrhs-xrhs': xrhs' = Subst xrhs Y (Arden Y yrhs)
 and X-in: (X, xrhs) \in ES **by** (simp add:Subst-all-def, blast)
 have rhss xrhs' \subseteq lhss ?B
 proof –
 have lhss ?B = lhss ES **by** (auto simp add:lhss-def Subst-all-def)
 moreover have rhss xrhs' \subseteq lhss ES
 proof –
 have rhss xrhs' \subseteq rhss xrhs \cup rhss (Arden Y yrhs) $- \{Y\}$
 proof –
 have Y \notin rhss (Arden Y yrhs)
 using Arden-removes-cl **by** auto
 thus ?thesis **using** xrhs-xrhs' **by** (auto simp: Subst-updates-cls)
 qed
 moreover have rhss xrhs \subseteq lhss ES \cup $\{Y\}$ **using** X-in sc

```

    apply (simp only: validity-def lhss-union-distrib)
    by (drule-tac x = (X, xrhs) in bspec, auto simp: lhss-def)
  moreover have rhss (Arden Y yrhs)  $\subseteq$  lhss ES  $\cup$  {Y}
    using sc
    by (auto simp add: Arden-removes-cl validity-def lhss-def)
  ultimately show ?thesis by auto
qed
ultimately show ?thesis by simp
qed
} thus ?thesis by (auto simp only: Subst-all-def validity-def)
qed

lemma Subst-all-satisfies-invariant:
  assumes invariant-ES: invariant (ES  $\cup$  {(Y, yrhs)})
  shows invariant (Subst-all ES Y (Arden Y yrhs))
proof (rule invariantI)
  have Y-eq-yrhs: Y = lang-rhs yrhs
    using invariant-ES by (simp only: invariant-def soundness-def, blast)
  have finite-yrhs: finite yrhs
    using invariant-ES by (auto simp: invariant-def finite-rhs-def)
  have ardenable-yrhs: ardenable yrhs
    using invariant-ES by (auto simp: invariant-def ardenable-all-def)
  show soundness (Subst-all ES Y (Arden Y yrhs))
  proof -
    have Y = lang-rhs (Arden Y yrhs)
      using Y-eq-yrhs invariant-ES finite-yrhs
      using finite-Trn[OF finite-yrhs]
      apply (rule-tac Arden-preserves-soundness)
      apply (simp-all)
      unfolding invariant-def ardenable-all-def ardenable-def
      apply (auto)
      done
    thus ?thesis using invariant-ES
      unfolding invariant-def finite-rhs-def2 soundness-def Subst-all-def
      by (auto simp add: Subst-preserves-soundness simp del: lang-rhs.simps)
  qed
  show finite (Subst-all ES Y (Arden Y yrhs))
    using invariant-ES by (simp add: invariant-def Subst-all-preserves-finite)
  show distinctness (Subst-all ES Y (Arden Y yrhs))
    using invariant-ES
    unfolding distinctness-def Subst-all-def invariant-def by auto
  show ardenable-all (Subst-all ES Y (Arden Y yrhs))
  proof -
    { fix X rhs
      assume (X, rhs)  $\in$  ES
      hence ardenable rhs using invariant-ES
        by (auto simp add: invariant-def ardenable-all-def)
      with ardenable-yrhs
      have ardenable (Subst rhs Y (Arden Y yrhs))

```

```

    by (simp add:ardenable-yrhs
        Subst-preserves-ardenable Arden-preserves-ardenable)
  } thus ?thesis by (auto simp add:ardenable-all-def Subst-all-def)
qed
show finite-rhs (Subst-all ES Y (Arden Y yrhs))
proof -
  have finite-rhs ES using invariant-ES
  by (simp add:invariant-def finite-rhs-def)
  moreover have finite (Arden Y yrhs)
  proof -
    have finite yrhs using invariant-ES
    by (auto simp:invariant-def finite-rhs-def)
    thus ?thesis using Arden-preserves-finite by auto
  qed
  ultimately show ?thesis
  by (simp add:Subst-all-preserves-finite-rhs)
qed
show validity (Subst-all ES Y (Arden Y yrhs))
  using invariant-ES Subst-all-preserves-validity by (auto simp add: invari-
ant-def)
qed

```

lemma *Remove-in-card-measure:*

```

  assumes finite: finite ES
  and in-ES: (X, rhs) ∈ ES
  shows (Remove ES X rhs, ES) ∈ measure card
proof -
  define f where f x = ((fst x)::'a lang, Subst (snd x) X (Arden X rhs)) for x
  define ES' where ES' = ES - {(X, rhs)}
  have Subst-all ES' X (Arden X rhs) = f ' ES'
  apply (auto simp: Subst-all-def f-def image-def)
  by (rule-tac x = (Y, yrhs) in bexI, simp+)
  then have card (Subst-all ES' X (Arden X rhs)) ≤ card ES'
  unfolding ES'-def using finite by (auto intro: card-image-le)
  also have ... < card ES unfolding ES'-def
  using in-ES finite by (rule-tac card-Diff1-less)
  finally show (Remove ES X rhs, ES) ∈ measure card
  unfolding Remove-def ES'-def by simp
qed

```

lemma *Subst-all-cls-remains:*

```

(X, xrhs) ∈ ES ⇒ ∃ xrhs'. (X, xrhs') ∈ (Subst-all ES Y yrhs)
by (auto simp: Subst-all-def)

```

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

```

  assumes card: Cond ES
  and e-in: (X, xrhs) ∈ ES
  and finite: finite ES

```

shows $\exists(Y, yrhs) \in ES. (X, xrhs) \neq (Y, yrhs)$
proof –
have $card\ ES > 1$ **using** $card\ e\text{-in}\ finite$
by $(cases\ card\ ES)\ (auto)$
then have $card\ (ES - \{(X, xrhs)\}) > 0$
using $finite\ e\text{-in}\ by\ auto$
then have $(ES - \{(X, xrhs)\}) \neq \{\}$ **using** $finite\ by\ (rule\text{-tac}\ notI,\ simp)$
then show $\exists(Y, yrhs) \in ES. (X, xrhs) \neq (Y, yrhs)$
by $auto$
qed

lemma *iteration-step-measure*:
assumes $Inv\text{-}ES: invariant\ ES$
and $X\text{-in}\text{-}ES: (X, xrhs) \in ES$
and $Cnd: Cond\ ES$
shows $(Iter\ X\ ES, ES) \in measure\ card$
proof –
have $fin: finite\ ES$ **using** $Inv\text{-}ES\ unfolding\ invariant\text{-}def\ by\ simp$
then obtain $Y\ yrhs$
where $Y\text{-in}\text{-}ES: (Y, yrhs) \in ES$ **and** $not\text{-}eq: (X, xrhs) \neq (Y, yrhs)$
using $Cnd\ X\text{-in}\text{-}ES\ by\ (drule\text{-tac}\ card\ noteq\ 1\ \text{-has}\ \text{-more})\ (auto)$
then have $(Y, yrhs) \in ES\ X \neq Y$
using $X\text{-in}\text{-}ES\ Inv\text{-}ES\ unfolding\ invariant\text{-}def\ distinctness\text{-}def$
by $auto$
then show $(Iter\ X\ ES, ES) \in measure\ card$
apply $(rule\ IterI2)$
apply $(rule\ Remove\text{-in}\text{-}card\ \text{-measure})$
apply $(simp\text{-all}\ add: fin)$
done
qed

lemma *iteration-step-invariant*:
assumes $Inv\text{-}ES: invariant\ ES$
and $X\text{-in}\text{-}ES: (X, xrhs) \in ES$
and $Cnd: Cond\ ES$
shows $invariant\ (Iter\ X\ ES)$
proof –
have $finite\text{-}ES: finite\ ES$ **using** $Inv\text{-}ES\ by\ (simp\ add: invariant\text{-}def)$
then obtain $Y\ yrhs$
where $Y\text{-in}\text{-}ES: (Y, yrhs) \in ES$ **and** $not\text{-}eq: (X, xrhs) \neq (Y, yrhs)$
using $Cnd\ X\text{-in}\text{-}ES\ by\ (drule\text{-tac}\ card\ noteq\ 1\ \text{-has}\ \text{-more})\ (auto)$
then have $(Y, yrhs) \in ES\ X \neq Y$
using $X\text{-in}\text{-}ES\ Inv\text{-}ES\ unfolding\ invariant\text{-}def\ distinctness\text{-}def$
by $auto$
then show $invariant\ (Iter\ X\ ES)$
proof $(rule\ IterI2)$
fix $Y\ yrhs$
assume $h: (Y, yrhs) \in ES\ X \neq Y$
then have $ES - \{(Y, yrhs)\} \cup \{(Y, yrhs)\} = ES$ **by** $auto$

```

    then show invariant (Remove ES Y yrhs) unfolding Remove-def
      using Inv-ES
      by (rule-tac Subst-all-satisfies-invariant) (simp)
  qed
qed

```

```

lemma iteration-step-ex:
  assumes Inv-ES: invariant ES
  and X-in-ES:  $(X, xrhs) \in ES$ 
  and Cnd: Cond ES
  shows  $\exists xrhs'. (X, xrhs') \in (Iter\ X\ ES)$ 
proof –
  have finite-ES: finite ES using Inv-ES by (simp add: invariant-def)
  then obtain Y yrhs
    where  $(Y, yrhs) \in ES$   $(X, xrhs) \neq (Y, yrhs)$ 
    using Cnd X-in-ES by (drule-tac card-noteq-1-has-more) (auto)
  then have  $(Y, yrhs) \in ES$   $X \neq Y$ 
    using X-in-ES Inv-ES unfolding invariant-def distinctness-def
    by auto
  then show  $\exists xrhs'. (X, xrhs') \in (Iter\ X\ ES)$ 
  apply(rule IterI2)
  unfolding Remove-def
  apply(rule Subst-all-cls-remains)
  using X-in-ES
  apply(auto)
  done
qed

```

2.7 The conclusion of the first direction

```

lemma Solve:
  fixes A::('a::finite) lang
  assumes fin: finite (UNIV // ≈A)
  and X-in:  $X \in (UNIV // ≈A)$ 
  shows  $\exists rhs. Solve\ X\ (Init\ (UNIV // ≈A)) = \{(X, rhs)\} \wedge invariant\ \{(X, rhs)\}$ 
proof –
  define Inv where Inv ES  $\longleftrightarrow invariant\ ES \wedge (\exists rhs. (X, rhs) \in ES)$  for ES
  have Inv (Init (UNIV // ≈A)) unfolding Inv-def
    using fin X-in by (simp add: Init-ES-satisfies-invariant, simp add: Init-def)
  moreover
  { fix ES
    assume inv: Inv ES and crd: Cond ES
    then have Inv (Iter X ES)
      unfolding Inv-def
      by (auto simp add: iteration-step-invariant iteration-step-ex) }
  moreover
  { fix ES
    assume inv: Inv ES and not-crd:  $\neg Cond\ ES$ 
    from inv obtain rhs where  $(X, rhs) \in ES$  unfolding Inv-def by auto

```

```

moreover
from not-crd have  $\text{card } ES = 1$  by simp
ultimately
have  $ES = \{(X, rhs)\}$  by (auto simp add: card-Suc-eq)
then have  $\exists rhs'. ES = \{(X, rhs')\} \wedge \text{invariant } \{(X, rhs')\}$  using inv
  unfolding Inv-def by auto }
moreover
  have wf (measure card) by simp
moreover
{ fix ES
  assume inv: Inv ES and crd: Cond ES
  then have  $(\text{Iter } X \text{ } ES, ES) \in \text{measure card}$ 
    unfolding Inv-def
    apply(clarify)
    apply(rule-tac iteration-step-measure)
    apply(auto)
    done }
ultimately
show  $\exists rhs. \text{Solve } X (\text{Init } (UNIV // \approx A)) = \{(X, rhs)\} \wedge \text{invariant } \{(X, rhs)\}$ 
  unfolding Solve-def by (rule while-rule)
qed

```

```

lemma every-egcl-has-reg:
  fixes A::('a::finite) lang
  assumes finite-CS: finite (UNIV // \approx A)
  and X-in-CS: X \in (UNIV // \approx A)
  shows  $\exists r. X = \text{lang } r$ 
proof –
  from finite-CS X-in-CS
  obtain xrhs where Inv-ES: invariant \{(X, xrhs)\}
    using Solve by metis

```

```

define A where  $A = \text{Arden } X \text{ } xrhs$ 
have  $\text{rhss } xrhs \subseteq \{X\}$  using Inv-ES
  unfolding validity-def invariant-def rhss-def lhss-def
  by auto
then have  $\text{rhss } A = \{X\}$  unfolding A-def
  by (simp add: Arden-removes-cl)
then have  $\text{eq: } \{\text{Lam } r \mid r. \text{Lam } r \in A\} = A$  unfolding rhss-def
  by (auto, case-tac x, auto)

```

```

have finite A using Inv-ES unfolding A-def invariant-def finite-rhs-def
  using Arden-preserves-finite by auto
then have fin: finite \{r. Lam r \in A\} by (rule finite-Lam)

```

```

have  $X = \text{lang-rhs } xrhs$  using Inv-ES unfolding invariant-def soundness-def
  by simp
then have  $X = \text{lang-rhs } A$  using Inv-ES
  unfolding A-def invariant-def ardenable-all-def finite-rhs-def

```

```

  by (rule-tac Arden-preserves-soundness) (simp-all add: finite-Trn)
  then have  $X = \text{lang-rhs } \{Lam\ r \mid r. Lam\ r \in A\}$  using eq by simp
  then have  $X = \text{lang } (\biguplus \{r. Lam\ r \in A\})$  using fin by auto
  then show  $\exists r. X = \text{lang } r$  by blast
qed

```

```

lemma bchoice-finite-set:
  assumes a:  $\forall x \in S. \exists y. x = f\ y$ 
  and b: finite S
  shows  $\exists ys. (\bigcup S) = \bigcup (f\ 'ys) \wedge \text{finite } ys$ 
using bchoice[OF a] b
apply (erule-tac exE)
apply (rule-tac x=fa ' S in exI)
apply (auto)
done

```

```

theorem Myhill-Nerode1:
  fixes A::('a::finite) lang
  assumes finite-CS: finite (UNIV //  $\approx$ A)
  shows  $\exists r. A = \text{lang } r$ 
proof -
  have fin: finite (finals A)
    using finals-in-partitions finite-CS by (rule finite-subset)
  have  $\forall X \in (UNIV // \approx A). \exists r. X = \text{lang } r$ 
    using finite-CS every-egcl-has-reg by blast
  then have a:  $\forall X \in \text{finals } A. \exists r. X = \text{lang } r$ 
    using finals-in-partitions by auto
  then obtain rs::('a rexp) set where  $\bigcup (\text{finals } A) = \bigcup (\text{lang } 'rs) \text{ finite } rs$ 
    using fin by (auto dest: bchoice-finite-set)
  then have  $A = \text{lang } (\biguplus rs)$ 
    unfolding lang-is-union-of-finals[symmetric] by simp
  then show  $\exists r. A = \text{lang } r$  by blast
qed

```

end

```

theory Myhill-2
  imports Myhill-1 HOL-Library.Sublist
begin

```

3 Second direction of MN: regular language \Rightarrow finite partition

3.1 Tagging functions

definition

```

  tag-eg :: ('a list  $\Rightarrow$  'b)  $\Rightarrow$  ('a list  $\times$  'a list) set (=)
where

```

$=tag= \equiv \{(x, y). tag\ x = tag\ y\}$

abbreviation

$tag\text{-}eq\text{-}applied :: 'a\ list \Rightarrow ('a\ list \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow bool\ (-\ ==\ -)$

where

$x\ =tag=\ y \equiv (x, y) \in =tag=$

lemma *[simp]*:

shows $(\approx A) \text{ `` } \{x\} = (\approx A) \text{ `` } \{y\} \longleftrightarrow x \approx A\ y$

unfolding *str-eq-def* **by** *auto*

lemma *refined-intro*:

assumes $\bigwedge x\ y\ z. \llbracket x\ =tag=\ y; x\ @\ z \in A \rrbracket \Longrightarrow y\ @\ z \in A$

shows $=tag= \subseteq \approx A$

using *assms* **unfolding** *str-eq-def* *tag-eq-def*

apply (*clarify*, *simp* (*no-asm-use*))

by *metis*

lemma *finite-eq-tag-rel*:

assumes *rng-fnt*: *finite* (*range* *tag*)

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

proof –

let $?f = \lambda X. tag\ ' X$ **and** $?A = (UNIV // =tag=)$

have *finite* ($?f\ ' ?A$)

proof –

have $range\ ?f \subseteq (Pow\ (range\ tag))$ **unfolding** *Pow-def* **by** *auto*

moreover

have *finite* ($Pow\ (range\ tag)$) **using** *rng-fnt* **by** *simp*

ultimately

have *finite* ($range\ ?f$) **unfolding** *image-def* **by** (*blast* *intro*: *finite-subset*)

moreover

have $?f\ ' ?A \subseteq range\ ?f$ **by** *auto*

ultimately **show** *finite* ($?f\ ' ?A$) **by** (*rule* *rev-finite-subset*)

qed

moreover

have *inj-on* $?f\ ?A$

proof –

{ **fix** $X\ Y$

assume *X-in*: $X \in ?A$

and *Y-in*: $Y \in ?A$

and *tag-eq*: $?f\ X = ?f\ Y$

then **obtain** $x\ y$

where $x \in X\ y \in Y\ tag\ x = tag\ y$

unfolding *quotient-def* *Image-def* *image-def* *tag-eq-def*

by (*simp*) (*blast*)

with *X-in* *Y-in*

have $X = Y$

unfolding *quotient-def* *tag-eq-def* **by** *auto*

}

then show *inj-on ?f ?A unfolding inj-on-def by auto*
qed
ultimately show *finite (UNIV // =tag=) by (rule finite-imageD)*
qed

lemma *refined-partition-finite:*

assumes *fnt: finite (UNIV // R1)*
and *refined: R1 \subseteq R2*
and *eq1: equiv UNIV R1 and eq2: equiv UNIV R2*
shows *finite (UNIV // R2)*

proof –

let *?f = $\lambda X. \{R1 \text{ “ } \{x\} \mid x. x \in X\}$*
and *?A = UNIV // R2 and ?B = UNIV // R1*
have *?f ‘ ?A \subseteq Pow ?B*
unfolding *image-def Pow-def quotient-def by auto*

moreover

have *finite (Pow ?B) using fnt by simp*

ultimately

have *finite (?f ‘ ?A) by (rule finite-subset)*

moreover

have *inj-on ?f ?A*

proof –

{ fix *X Y*
assume *X-in: X \in ?A and Y-in: Y \in ?A and eq-f: ?f X = ?f Y*
from *quotientE [OF X-in]*
obtain *x where X = R2 “ {x} by blast*
with *equiv-class-self[OF eq2] have x-in: x \in X by simp*
then have *R1 “ {x} \in ?f X by auto*
with *eq-f have R1 “ {x} \in ?f Y by simp*
then obtain *y*
where *y-in: y \in Y and eq-r1-xy: R1 “ {x} = R1 “ {y} by auto*
with *eq-equiv-class[OF - eq1]*
have *(x, y) \in R1 by blast*
with *refined have (x, y) \in R2 by auto*
with *quotient-eqI [OF eq2 X-in Y-in x-in y-in]*
have *X = Y .*

}

then show *inj-on ?f ?A unfolding inj-on-def by blast*

qed

ultimately show *finite (UNIV // R2) by (rule finite-imageD)*

qed

lemma *tag-finite-imageD:*

assumes *rng-fnt: finite (range tag)*

and *refined: =tag= \subseteq \approx A*

shows *finite (UNIV // \approx A)*

proof (*rule-tac refined-partition-finite [of =tag=]*)

show *finite (UNIV // =tag=) by (rule finite-eq-tag-rel[OF rng-fnt])*

next

```

  show =tag=  $\subseteq \approx A$  using refined .
next
  show equiv UNIV =tag=
  and equiv UNIV ( $\approx A$ )
    unfolding equiv-def str-eq-def tag-eq-def refl-on-def sym-def trans-def
    by auto
qed

```

3.2 Base cases: Zero, One and Atom

```

lemma quot-zero-eq:
  shows UNIV //  $\approx \{\}$  = {UNIV}
unfolding quotient-def Image-def str-eq-def by auto

```

```

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

```

```

lemma quot-one-subset:
  shows UNIV //  $\approx \{\}$   $\subseteq \{\{\}\}$ , UNIV -  $\{\{\}\}$ 
proof
  fix x
  assume x  $\in$  UNIV //  $\approx \{\}$ 
  then obtain y where h: x = {z. y  $\approx \{\}$  z}
    unfolding quotient-def Image-def by blast
  { assume y =  $\{\}$ 
    with h have x =  $\{\{\}\}$  by (auto simp: str-eq-def)
    then have x  $\in \{\{\}\}$ , UNIV -  $\{\{\}\}$  by simp }
  moreover
  { assume y  $\neq \{\}$ 
    with h have x = UNIV -  $\{\{\}\}$  by (auto simp: str-eq-def)
    then have x  $\in \{\{\}\}$ , UNIV -  $\{\{\}\}$  by simp }
  ultimately show x  $\in \{\{\}\}$ , UNIV -  $\{\{\}\}$  by blast
qed

```

```

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

```

```

lemma quot-atom-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-def*) }
moreover
 { **assume** $y = [c]$ **hence** $x = \{[c]\}$ **using** h
 by (*auto dest!: spec[where $x = []$ simp: str-eq-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-def*)
 }
ultimately show *?thesis* **by** *blast*
qed
qed

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

3.3 Case for *Plus*

definition

tag-Plus :: 'a lang \Rightarrow 'a lang \Rightarrow 'a list \Rightarrow ('a lang \times 'a lang)
where
tag-Plus $A B \equiv \lambda x. (\approx A \text{ `` } \{x\}, \approx B \text{ `` } \{x\})$

lemma *quot-plus-finiteI* [*intro*]:
 assumes *finite1*: *finite* ($UNIV // \approx A$)
 and *finite2*: *finite* ($UNIV // \approx B$)
 shows *finite* ($UNIV // \approx(A \cup B)$)
proof (*rule-tac tag = tag-Plus A B in tag-finite-imageD*)
 have *finite* (($UNIV // \approx A$) \times ($UNIV // \approx B$))
 using *finite1 finite2* **by** *auto*
 then show *finite* (*range* (*tag-Plus A B*))
 unfolding *tag-Plus-def quotient-def*
 by (*rule rev-finite-subset*) (*auto*)
next
 show $=tag-Plus A B = \subseteq \approx(A \cup B)$
 unfolding *tag-eq-def tag-Plus-def str-eq-def* **by** *auto*
qed

3.4 Case for *Times*

definition

Partitions $x \equiv \{(x_p, x_s). x_p @ x_s = x\}$

lemma *conc-partitions-elim*:

assumes $x \in A \cdot B$
shows $\exists (u, v) \in \text{Partitions } x. u \in A \wedge v \in B$
using *assms unfolding conc-def Partitions-def*
by *auto*

lemma *conc-partitions-intro*:
assumes $(u, v) \in \text{Partitions } x \wedge u \in A \wedge v \in B$
shows $x \in A \cdot B$
using *assms unfolding conc-def Partitions-def*
by *auto*

lemma *equiv-class-member*:
assumes $x \in A$
and $\approx A \text{ `` } \{x\} = \approx A \text{ `` } \{y\}$
shows $y \in A$
using *assms*
apply(*simp*)
apply(*simp add: str-eq-def*)
apply(*metis append-Nil2*)
done

definition

tag-Times :: 'a lang \Rightarrow 'a lang \Rightarrow 'a list \Rightarrow 'a lang \times 'a lang set
where
tag-Times A B $\equiv \lambda x. (\approx A \text{ `` } \{x\}, \{(\approx B \text{ `` } \{x_s\}) \mid x_p \ x_s. x_p \in A \wedge (x_p, x_s) \in \text{Partitions } x\})$

lemma *tag-Times-injI*:
assumes $a: \text{tag-Times } A \ B \ x = \text{tag-Times } A \ B \ y$
and $c: x \ @ \ z \in A \cdot B$
shows $y \ @ \ z \in A \cdot B$
proof –
from c **obtain** $u \ v$ **where**
 $h1: (u, v) \in \text{Partitions } (x \ @ \ z)$ **and**
 $h2: u \in A$ **and**
 $h3: v \in B$ **by** (*auto dest: conc-partitions-elim*)
from $h1$ **have** $x \ @ \ z = u \ @ \ v$ **unfolding** *Partitions-def* **by** *simp*
then obtain us
where $(x = u \ @ \ us \wedge us \ @ \ z = v) \vee (x \ @ \ us = u \wedge z = us \ @ \ v)$
by (*auto simp add: append-eq-append-conv2*)
moreover
{ **assume** $eq: x = u \ @ \ us \ us \ @ \ z = v$
have $(\approx B \text{ `` } \{us\}) \in \text{snd } (\text{tag-Times } A \ B \ x)$
unfolding *Partitions-def tag-Times-def* **using** $h2 \ eq$
by (*auto simp add: str-eq-def*)
then have $(\approx B \text{ `` } \{us\}) \in \text{snd } (\text{tag-Times } A \ B \ y)$
using a **by** *simp*
then obtain $u' \ us'$ **where**
 $q1: u' \in A$ **and**

```

    q2:  $\approx B$  “  $\{us\} = \approx B$  “  $\{us'\}$  and
    q3:  $(u', us') \in \text{Partitions } y$ 
    unfolding tag-Times-def by auto
  from q2 h3 eq
  have  $us' @ z \in B$ 
    unfolding Image-def str-eq-def by auto
  then have  $y @ z \in A \cdot B$  using q1 q3
    unfolding Partitions-def by auto
}
moreover
{ assume eq:  $x @ us = u z = us @ v$ 
  have  $(\approx A$  “  $\{x\}) = \text{fst } (\text{tag-Times } A B x)$ 
    by (simp add: tag-Times-def)
  then have  $(\approx A$  “  $\{x\}) = \text{fst } (\text{tag-Times } A B y)$ 
    using a by simp
  then have  $\approx A$  “  $\{x\} = \approx A$  “  $\{y\}$ 
    by (simp add: tag-Times-def)
  moreover
  have  $x @ us \in A$  using h2 eq by simp
  ultimately
  have  $y @ us \in A$  using equiv-class-member
    unfolding Image-def str-eq-def by blast
  then have  $(y @ us) @ v \in A \cdot B$ 
    using h3 unfolding conc-def by blast
  then have  $y @ z \in A \cdot B$  using eq by simp
}
ultimately show  $y @ z \in A \cdot B$  by blast
qed

lemma quot-conc-finiteI [intro]:
  assumes fin1: finite (UNIV //  $\approx A$ )
  and fin2: finite (UNIV //  $\approx B$ )
  shows finite (UNIV //  $\approx(A \cdot B)$ )
proof (rule-tac tag = tag-Times A B in tag-finite-imageD)
  have  $\bigwedge x y z. \llbracket \text{tag-Times } A B x = \text{tag-Times } A B y; x @ z \in A \cdot B \rrbracket \implies y @ z \in A \cdot B$ 
    by (rule tag-Times-injI)
      (auto simp add: tag-Times-def tag-eq-def)
  then show  $=\text{tag-Times } A B = \subseteq \approx(A \cdot B)$ 
    by (rule refined-intro)
      (auto simp add: tag-eq-def)
next
  have *: finite ((UNIV //  $\approx A$ )  $\times$  (Pow (UNIV //  $\approx B$ )))
    using fin1 fin2 by auto
  show finite (range (tag-Times A B))
    unfolding tag-Times-def
    apply(rule finite-subset[OF - *])
    unfolding quotient-def
    by auto

```

qed

3.5 Case for *Star*

lemma *star-partitions-elim*:

assumes $x @ z \in A^\star$ $x \neq []$

shows $\exists (u, v) \in \text{Partitions } (x @ z). \text{strict-prefix } u \ x \wedge u \in A^\star \wedge v \in A^\star$

proof –

have $([], x @ z) \in \text{Partitions } (x @ z)$ *strict-prefix* $[] \ x \ [] \in A^\star$ $x @ z \in A^\star$

using *assms* **by** (*auto simp add: Partitions-def strict-prefix-def*)

then show $\exists (u, v) \in \text{Partitions } (x @ z). \text{strict-prefix } u \ x \wedge u \in A^\star \wedge v \in A^\star$

by *blast*

qed

lemma *finite-set-has-max2*:

$\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \exists \text{max} \in A. \forall a \in A. \text{length } a \leq \text{length } \text{max}$

apply (*induct rule:finite.induct*)

apply (*simp*)

by (*metis (no-types) all-not-in-conv insert-iff linorder-le-cases order-trans*)

lemma *finite-strict-prefix-set*:

shows *finite* $\{x a. \text{strict-prefix } x a \ (x::'a \text{ list})\}$

apply (*induct x rule:rev-induct, simp*)

apply (*subgoal-tac* $\{x a. \text{strict-prefix } x a \ (x s @ [x])\} = \{x a. \text{strict-prefix } x a \ x s\} \cup \{x s\}$)

by (*auto simp:strict-prefix-def*)

lemma *append-eq-cases*:

assumes $a: x @ y = m @ n$ $m \neq []$

shows *prefix* $x \ m \vee \text{strict-prefix } m \ x$

unfolding *prefix-def strict-prefix-def* **using** a

by (*auto simp add: append-eq-append-conv2*)

lemma *star-spartitions-elim2*:

assumes $a: x @ z \in A^\star$

and $b: x \neq []$

shows $\exists (u, v) \in \text{Partitions } x. \exists (u', v') \in \text{Partitions } z. \text{strict-prefix } u \ x \wedge u \in A^\star \wedge v @ u' \in A \wedge v' \in A^\star$

proof –

define S **where** $S = \{u \mid u \ v. (u, v) \in \text{Partitions } x \wedge \text{strict-prefix } u \ x \wedge u \in A^\star \wedge v @ z \in A^\star\}$

have *finite* $\{u. \text{strict-prefix } u \ x\}$ **by** (*rule finite-strict-prefix-set*)

then have *finite* S **unfolding** $S\text{-def}$

by (*rule rev-finite-subset*) (*auto*)

moreover

have $S \neq \{\}$ **using** $a \ b$ **unfolding** $S\text{-def Partitions-def}$

by (*auto simp: strict-prefix-def*)

ultimately have $\exists u\text{-max} \in S. \forall u \in S. \text{length } u \leq \text{length } u\text{-max}$

using *finite-set-has-max2* **by** *blast*

then obtain $u\text{-max } v$
where $h0: (u\text{-max}, v) \in \text{Partitions } x$
and $h1: \text{strict-prefix } u\text{-max } x$
and $h2: u\text{-max} \in A^\star$
and $h3: v @ z \in A^\star$
and $h4: \forall u v. (u, v) \in \text{Partitions } x \wedge \text{strict-prefix } u x \wedge u \in A^\star \wedge v @ z \in A^\star \longrightarrow \text{length } u \leq \text{length } u\text{-max}$
unfolding $S\text{-def Partitions-def}$ **by** blast
have $q: v \neq []$ **using** $h0 h1 b$ **unfolding** Partitions-def **by** auto
from $h3$ **obtain** $a b$
where $i1: (a, b) \in \text{Partitions } (v @ z)$
and $i2: a \in A$
and $i3: b \in A^\star$
and $i4: a \neq []$
unfolding Partitions-def
using q **by** $(\text{auto dest: star-decom})$
have $\text{prefix } v a$
proof (rule ccontr)
assume $a: \neg(\text{prefix } v a)$
from $i1$ **have** $i1': a @ b = v @ z$ **unfolding** Partitions-def **by** simp
then have $\text{prefix } a v \vee \text{strict-prefix } v a$ **using** $\text{append-eq-cases } q$ **by** blast
then have $q: \text{strict-prefix } a v$ **using** a **unfolding** $\text{strict-prefix-def prefix-def}$ **by** auto
then obtain as **where** $eq: a @ as = v$ **unfolding** $\text{strict-prefix-def prefix-def}$ **by** auto
have $(u\text{-max} @ a, as) \in \text{Partitions } x$ **using** $eq h0$ **unfolding** Partitions-def **by** auto
moreover
have $\text{strict-prefix } (u\text{-max} @ a) x$ **using** $h0 eq q$ **unfolding** $\text{Partitions-def prefix-def strict-prefix-def}$ **by** auto
moreover
have $u\text{-max} @ a \in A^\star$ **using** $i2 h2$ **by** simp
moreover
have $as @ z \in A^\star$ **using** $i1' i2 i3 eq$ **by** auto
ultimately have $\text{length } (u\text{-max} @ a) \leq \text{length } u\text{-max}$ **using** $h4$ **by** blast
with $i4$ **show** False **by** auto
qed
with $i1$ **obtain** $za zb$
where $k1: v @ za = a$
and $k2: (za, zb) \in \text{Partitions } z$
and $k4: zb = b$
unfolding $\text{Partitions-def prefix-def}$
by $(\text{auto simp add: append-eq-append-conv2})$
show $\exists (u, v) \in \text{Partitions } x. \exists (u', v') \in \text{Partitions } z. \text{strict-prefix } u x \wedge u \in A^\star \wedge v @ u' \in A \wedge v' \in A^\star$
using $h0 h1 h2 i2 i3 k1 k2 k4$ **unfolding** Partitions-def **by** blast
qed

definition

tag-Star :: 'a lang \Rightarrow 'a list \Rightarrow ('a lang) set
where
tag-Star A $\equiv \lambda x. \{ \approx A \text{ `` } \{v\} \mid u v. \text{strict-prefix } u x \wedge u \in A^\star \wedge (u, v) \in \text{Partitions } x \}$

lemma *tag-Star-non-empty-injI*:

assumes *a*: *tag-Star* A *x* = *tag-Star* A *y*
and *c*: *x* @ *z* $\in A^\star$
and *d*: *x* $\neq []$
shows *y* @ *z* $\in A^\star$
proof –
obtain *u v u' v'*
where *a1*: (*u*, *v*) $\in \text{Partitions } x$ (*u'*, *v'*) $\in \text{Partitions } z$
and *a2*: *strict-prefix* *u x*
and *a3*: *u* $\in A^\star$
and *a4*: *v* @ *u'* $\in A$
and *a5*: *v'* $\in A^\star$
using *c d* **by** (*auto dest: star-spartitions-elim2*)
have ($\approx A$) `` $\{v\} \in \text{tag-Star } A \ x$
apply(*simp add: tag-Star-def Partitions-def str-eq-def*)
using *a1 a2 a3* **by** (*auto simp add: Partitions-def*)
then have ($\approx A$) `` $\{v\} \in \text{tag-Star } A \ y$ **using** *a* **by** *simp*
then obtain *u1 v1*
where *b1*: *v* $\approx A \ v1$
and *b3*: *u1* $\in A^\star$
and *b4*: (*u1*, *v1*) $\in \text{Partitions } y$
unfolding *tag-Star-def* **by** *auto*
have *c*: *v1* @ *u'* $\in A^\star$ **using** *b1 a4* **unfolding** *str-eq-def* **by** *simp*
have *u1* @ (*v1* @ *u'*) @ *v'* $\in A^\star$
using *b3 c a5* **by** (*simp only: append-in-starI*)
then show *y* @ *z* $\in A^\star$ **using** *b4 a1*
unfolding *Partitions-def* **by** *auto*
qed

lemma *tag-Star-empty-injI*:

assumes *a*: *tag-Star* A *x* = *tag-Star* A *y*
and *c*: *x* @ *z* $\in A^\star$
and *d*: *x* = []
shows *y* @ *z* $\in A^\star$
proof –
from *a* **have** {} = *tag-Star* A *y* **unfolding** *tag-Star-def* **using** *d* **by** *auto*
then have *y* = []
unfolding *tag-Star-def Partitions-def strict-prefix-def prefix-def*
by (*auto*) (*metis Nil-in-star append-self-conv2*)
then show *y* @ *z* $\in A^\star$ **using** *c d* **by** *simp*
qed

lemma *quot-star-finiteI* [*intro*]:

assumes *finite1*: *finite* (UNIV // $\approx A$)

```

shows finite (UNIV //  $\approx(A\star)$ )
proof (rule-tac tag = tag-Star A in tag-finite-imageD)
  have  $\bigwedge x y z. \llbracket \text{tag-Star } A \ x = \text{tag-Star } A \ y; x @ z \in A\star \rrbracket \implies y @ z \in A\star$ 
  by (case-tac x = []) (blast intro: tag-Star-empty-injI tag-Star-non-empty-injI)
  then show  $(\text{tag-Star } A) = \subseteq \approx(A\star)$ 
  by (rule refined-intro) (auto simp add: tag-eq-def)
next
  have *: finite (Pow (UNIV //  $\approx A$ ))
  using finite1 by auto
  show finite (range (tag-Star A))
  unfolding tag-Star-def
  by (rule finite-subset[OF - *])
  (auto simp add: quotient-def)
qed

```

3.6 The conclusion of the second direction

```

lemma Myhill-Nerode2:
  fixes r::'a rexp
  shows finite (UNIV //  $\approx(\text{lang } r)$ )
by (induct r) (auto)

end

```

```

theory Myhill
  imports Myhill-2 Regular-Sets.Derivatives
begin

```

4 The theorem

```

theorem Myhill-Nerode:
  fixes A::('a::finite) lang
  shows  $(\exists r. A = \text{lang } r) \longleftrightarrow \text{finite } (UNIV // \approx A)$ 
using Myhill-Nerode1 Myhill-Nerode2 by auto

```

4.1 Second direction proved using partial derivatives

An alternative proof using the notion of partial derivatives for regular expressions due to Antimirov [1].

```

lemma MN-Rel-Derivs:
  shows  $x \approx A \ y \longleftrightarrow \text{Derivs } x \ A = \text{Derivs } y \ A$ 
unfolding Derivs-def str-eq-def
by auto

```

```

lemma Myhill-Nerode3:
  fixes r::'a rexp
  shows finite (UNIV //  $\approx(\text{lang } r)$ )

```

```

proof –
  have finite (UNIV //  $=(\lambda x. pderivs\ x\ r)=$ )
  proof –
    have range ( $\lambda x. pderivs\ x\ r$ )  $\subseteq Pow$  (pderivs-lang UNIV r)
      unfolding pderivs-lang-def by auto
    moreover
      have finite (Pow (pderivs-lang UNIV r)) by (simp add: finite-pderivs-lang)
    ultimately
      have finite (range ( $\lambda x. pderivs\ x\ r$ ))
        by (simp add: finite-subset)
      then show finite (UNIV //  $=(\lambda x. pderivs\ x\ r)=$ )
        by (rule finite-eq-tag-rel)
    qed
  moreover
    have  $=(\lambda x. pderivs\ x\ r)= \subseteq \approx(lang\ r)$ 
      unfolding tag-eq-def
      by (auto simp add: MN-Rel-Derivs-Derivs-pderivs)
    moreover
      have equiv UNIV  $=(\lambda x. pderivs\ x\ r)=$ 
      and equiv UNIV  $\approx(lang\ r)$ 
        unfolding equiv-def refl-on-def sym-def trans-def
        unfolding tag-eq-def str-eq-def
        by auto
      ultimately show finite (UNIV //  $\approx(lang\ r)$ )
        by (rule refined-partition-finite)
    qed
  end

theory Closures
imports Myhill HOL-Library.Infinite-Set
begin

```

5 Closure properties of regular languages

abbreviation

regular :: 'a lang \Rightarrow bool

where

regular *A* $\equiv \exists r. A = lang\ r$

5.1 Closure under \cup , \cdot and \star

lemma *closure-union* [*intro*]:

assumes *regular* *A* *regular* *B*

shows *regular* (*A* \cup *B*)

proof –

from *assms* **obtain** *r1* *r2*::'a *rexpr* **where** *lang* *r1* = *A* *lang* *r2* = *B* **by** *auto*

then have *A* \cup *B* = *lang* (*Plus* *r1* *r2*) **by** *simp*

then show *regular* (*A* \cup *B*) **by** *blast*

qed

lemma *closure-seq* [intro]:

assumes *regular A regular B*

shows *regular (A · B)*

proof –

from *assms* obtain $r1\ r2::'a\ \text{rexp}$ where $\text{lang } r1 = A\ \text{lang } r2 = B$ by *auto*

then have $A \cdot B = \text{lang } (\text{Times } r1\ r2)$ by *simp*

then show *regular (A · B)* by *blast*

qed

lemma *closure-star* [intro]:

assumes *regular A*

shows *regular (A \star)*

proof –

from *assms* obtain $r::'a\ \text{rexp}$ where $\text{lang } r = A$ by *auto*

then have $A\star = \text{lang } (\text{Star } r)$ by *simp*

then show *regular (A \star)* by *blast*

qed

5.2 Closure under complementation

Closure under complementation is proved via the Myhill-Nerode theorem

lemma *closure-complement* [intro]:

fixes $A::('a::\text{finite})\ \text{lang}$

assumes *regular A*

shows *regular (– A)*

proof –

from *assms* have $\text{finite } (\text{UNIV} // \approx A)$ by (*simp add: Myhill-Nerode*)

then have $\text{finite } (\text{UNIV} // \approx (-A))$ by (*simp add: str-eq-def*)

then show *regular (– A)* by (*simp add: Myhill-Nerode*)

qed

5.3 Closure under – and \cap

lemma *closure-difference* [intro]:

fixes $A::('a::\text{finite})\ \text{lang}$

assumes *regular A regular B*

shows *regular (A – B)*

proof –

have $A - B = - (- A \cup B)$ by *blast*

moreover

have *regular (– (– A \cup B))*

using *assms* by *blast*

ultimately show *regular (A – B)* by *simp*

qed

lemma *closure-intersection* [intro]:

fixes $A::('a::\text{finite})\ \text{lang}$

```

assumes regular A regular B
shows regular (A ∩ B)
proof -
  have  $A \cap B = - (- A \cup - B)$  by blast
  moreover
  have regular (- (- A ∪ - B))
    using assms by blast
  ultimately show regular (A ∩ B) by simp
qed

```

5.4 Closure under string reversal

```

fun
  Rev :: 'a rexp ⇒ 'a rexp
where
  Rev Zero = Zero
| Rev One = One
| Rev (Atom c) = Atom c
| Rev (Plus r1 r2) = Plus (Rev r1) (Rev r2)
| Rev (Times r1 r2) = Times (Rev r2) (Rev r1)
| Rev (Star r) = Star (Rev r)

```

```

lemma rev-seq[simp]:
  shows  $\text{rev } (B \cdot A) = (\text{rev } A) \cdot (\text{rev } B)$ 
unfolding conc-def image-def
by (auto) (metis rev-append)+

```

```

lemma rev-star1:
  assumes  $a: s \in (\text{rev } A)^\star$ 
  shows  $s \in \text{rev } (A^\star)$ 
using a
proof(induct rule: star-induct)
  case (append s1 s2)
  have inj: inj (rev::'a list ⇒ 'a list) unfolding inj-on-def by auto
  have  $s1 \in \text{rev } A \ s2 \in \text{rev } (A^\star)$  by fact+
  then obtain  $x1 \ x2$  where  $x1 \in A \ x2 \in A^\star$  and eqs: s1 = rev x1 s2 = rev x2
by auto
  then have  $x1 \in A^\star \ x2 \in A^\star$  by (auto)
  then have  $x2 @ x1 \in A^\star$  by (auto)
  then have  $\text{rev } (x2 @ x1) \in \text{rev } A^\star$  using inj by (simp only: inj-image-mem-iff)
  then show  $s1 @ s2 \in \text{rev } A^\star$  using eqs by simp
qed (auto)

```

```

lemma rev-star2:
  assumes  $a: s \in A^\star$ 
  shows  $\text{rev } s \in (\text{rev } A)^\star$ 
using a
proof(induct rule: star-induct)
  case (append s1 s2)

```

have *inj*: *inj* (*rev*::'a list \Rightarrow 'a list) **unfolding** *inj-on-def* **by** *auto*
have *s1* \in *A* **by** *fact*
then have *rev s1* \in *rev* ' *A* **using** *inj* **by** (*simp only: inj-image-mem-iff*)
then have *rev s1* \in (*rev* ' *A*) \star **by** (*auto*)
moreover
have *rev s2* \in (*rev* ' *A*) \star **by** *fact*
ultimately show *rev (s1 @ s2)* \in (*rev* ' *A*) \star **by** (*auto*)
qed (*auto*)

lemma *rev-star* [*simp*]:
shows *rev* ' (*A* \star) = (*rev* ' *A*) \star
using *rev-star1* *rev-star2* **by** *auto*

lemma *rev-lang*:
shows *rev* ' (*lang r*) = *lang* (*Rev r*)
by (*induct r*) (*simp-all add: image-Un*)

lemma *closure-reversal* [*intro*]:
assumes *regular A*
shows *regular (rev* ' *A*)
proof –
from *assms* **obtain** *r*::'a *rexp* **where** *A* = *lang r* **by** *auto*
then have *lang (Rev r)* = *rev* ' *A* **by** (*simp add: rev-lang*)
then show *regular (rev* ' *A*) **by** *blast*
qed

5.5 Closure under left-quotients

abbreviation
Deriv-lang A B \equiv $\bigcup x \in A. \text{Derivs } x B$

lemma *closure-left-quotient*:
assumes *regular A*
shows *regular (Deriv-lang B A)*
proof –
from *assms* **obtain** *r*::'a *rexp* **where** *eq: lang r = A* **by** *auto*
have *fin*: *finite (pderivs-lang B r)* **by** (*rule finite-pderivs-lang*)

have *Deriv-lang B (lang r)* = (\bigcup (*lang* ' *pderivs-lang B r*))
by (*simp add: Derivs-pderivs pderivs-lang-def*)
also have \dots = *lang* (\biguplus (*pderivs-lang B r*)) **using** *fin* **by** *simp*
finally have *Deriv-lang B A* = *lang* (\biguplus (*pderivs-lang B r*)) **using** *eq*
by *simp*
then show *regular (Deriv-lang B A)* **by** *auto*
qed

5.6 Finite and co-finite sets are regular

lemma *singleton-regular*:
shows *regular* {*s*}

```

proof (induct s)
  case Nil
    have  $\{\emptyset\} = \text{lang } (\text{One})$  by simp
    then show regular  $\{\emptyset\}$  by blast
  next
    case (Cons c s)
    have regular  $\{s\}$  by fact
    then obtain r where  $\{s\} = \text{lang } r$  by blast
    then have  $\{c \# s\} = \text{lang } (\text{Times } (\text{Atom } c) r)$ 
      by (auto simp add: conc-def)
    then show regular  $\{c \# s\}$  by blast
qed

```

```

lemma finite-regular:
  assumes finite A
  shows regular A
using assms
proof (induct)
  case empty
    have  $\{\} = \text{lang } (\text{Zero})$  by simp
    then show regular  $\{\}$  by blast
  next
    case (insert s A)
    have regular  $\{s\}$  by (simp add: singleton-regular)
    moreover
    have regular A by fact
    ultimately have regular  $(\{s\} \cup A)$  by (rule closure-union)
    then show regular (insert s A) by simp
qed

```

```

lemma cofinite-regular:
  fixes A::'a::finite lang
  assumes finite  $(- A)$ 
  shows regular A
proof -
  from assms have regular  $(- A)$  by (simp add: finite-regular)
  then have regular  $(-(- A))$  by (rule closure-complement)
  then show regular A by simp
qed

```

5.7 Continuation lemma for showing non-regularity of languages

```

lemma continuation-lemma:
  fixes A B::'a::finite lang
  assumes reg: regular A
  and inf: infinite B
  shows  $\exists x \in B. \exists y \in B. x \neq y \wedge x \approx_A y$ 
proof -

```

```

define eqfun where eqfun = (λA x::('a::finite list). (≈A) “ {x}”)
have finite (UNIV // ≈A) using reg by (simp add: Myhill-Nerode)
moreover
have (eqfun A) ‘ B ⊆ UNIV // (≈A)
  unfolding eqfun-def quotient-def by auto
ultimately have finite ((eqfun A) ‘ B) by (rule rev-finite-subset)
with inf have ∃ a ∈ B. infinite {b ∈ B. eqfun A b = eqfun A a}
  by (rule pigeonhole-infinite)
then obtain a where in-a: a ∈ B and infinite {b ∈ B. eqfun A b = eqfun A a}
  by blast
moreover
have {b ∈ B. eqfun A b = eqfun A a} = {b ∈ B. b ≈A a}
  unfolding eqfun-def Image-def str-eq-def by auto
ultimately have infinite {b ∈ B. b ≈A a} by simp
then have infinite ({b ∈ B. b ≈A a} - {a}) by simp
moreover
have {b ∈ B. b ≈A a} - {a} = {b ∈ B. b ≈A a ∧ b ≠ a} by auto
ultimately have infinite {b ∈ B. b ≈A a ∧ b ≠ a} by simp
then have {b ∈ B. b ≈A a ∧ b ≠ a} ≠ {}
  by (metis finite.emptyI)
then obtain b where b ∈ B b ≠ a b ≈A a by blast
with in-a show ∃ x ∈ B. ∃ y ∈ B. x ≠ y ∧ x ≈A y
  by blast
qed

```

5.8 The language $a^n b^n$ is not regular

abbreviation

replicate-rev (- $\overset{\sim}{\sim}$ - [100, 100] 100)

where

$a \overset{\sim}{\sim} n \equiv \text{replicate } n \ a$

lemma *an-bn-not-regular*:

shows $\neg \text{regular } (\bigcup n. \{ \text{CHR } "a" \overset{\sim}{\sim} n @ \text{CHR } "b" \overset{\sim}{\sim} n \})$

proof

define A **where** A = $(\bigcup n. \{ \text{CHR } "a" \overset{\sim}{\sim} n @ \text{CHR } "b" \overset{\sim}{\sim} n \})$

assume as: regular A

define B **where** B = $(\bigcup n. \{ \text{CHR } "a" \overset{\sim}{\sim} n \})$

have sameness: $\bigwedge i \ j. \text{CHR } "a" \overset{\sim}{\sim} i @ \text{CHR } "b" \overset{\sim}{\sim} j \in A \longleftrightarrow i = j$

unfolding A-def

apply auto

apply(*drule-tac* f=λs. length (filter ((=) (CHR "a")) s) = length (filter ((=) (CHR "b")) s)

in arg-cong)

apply(simp)

done

have b: infinite B

```

  unfolding infinite-iff-countable-subset
  unfolding inj-on-def B-def
  by (rule-tac x= $\lambda n. CHR "a" \sim n$  in exI) (auto)
moreover
have  $\forall x \in B. \forall y \in B. x \neq y \longrightarrow \neg (x \approx_A y)$ 
  apply (auto)
  unfolding B-def
  apply (auto)
  apply (simp add: str-eq-def)
  apply (rule-tac x= $CHR "b" \sim xa$  in spec)
  apply (simp add: sameness)
done
ultimately
show False using continuation-lemma[OF as] by blast
qed

```

```

end
theory Closures2
imports
  Closures
  Well-Quasi-Orders.Well-Quasi-Orders
begin

```

6 Closure under *SUBSEQ* and *SUPSEQ*

Properties about the embedding relation

```

lemma subseq-strict-length:
  assumes a: subseq x y  $x \neq y$ 
  shows length x < length y
using a
by (induct) (auto simp add: less-Suc-eq)

```

```

lemma subseq-wf:
  shows wf  $\{(x, y). subseq x y \wedge x \neq y\}$ 
proof -
  have wf (measure length) by simp
  moreover
  have  $\{(x, y). subseq x y \wedge x \neq y\} \subseteq measure\ length$ 
    unfolding measure-def by (auto simp add: subseq-strict-length)
  ultimately
  show wf  $\{(x, y). subseq x y \wedge x \neq y\}$  by (rule wf-subset)
qed

```

```

lemma subseq-good:
  shows good subseq  $(f :: nat \Rightarrow ('a::finite) list)$ 
using wqo-on-imp-good[where f=f, OF wqo-on-lists-over-finite-sets]
by simp

```

lemma *subseq-Higman-antichains*:
assumes $a: \forall (x::('a::\text{finite}) \text{ list}) \in A. \forall y \in A. x \neq y \longrightarrow \neg(\text{subseq } x \ y) \wedge \neg(\text{subseq } y \ x)$
shows *finite A*
proof (*rule ccontr*)
assume *infinite A*
then obtain $f::\text{nat} \Rightarrow 'a::\text{finite list}$ **where** $b: \text{inj } f$ **and** $c: \text{range } f \subseteq A$
by (*auto simp add: infinite-iff-countable-subset*)
from *subseq-good*[**where** $f=f$]
obtain $i \ j$ **where** $d: i < j$ **and** $e: \text{subseq } (f \ i) \ (f \ j) \vee f \ i = f \ j$
unfolding *good-def*
by *auto*
have $f \ i \neq f \ j$ **using** $b \ d$ **by** (*auto simp add: inj-on-def*)
moreover
have $f \ i \in A$ **using** c **by** *auto*
moreover
have $f \ j \in A$ **using** c **by** *auto*
ultimately have $\neg(\text{subseq } (f \ i) \ (f \ j))$ **using** a **by** *simp*
with e **show** *False* **by** *auto*
qed

6.1 Sub- and Supersequences

definition

$SUBSEQ \ A \equiv \{x::('a::\text{finite}) \text{ list}. \exists y \in A. \text{subseq } x \ y\}$

definition

$SUPSEQ \ A \equiv \{x::('a::\text{finite}) \text{ list}. \exists y \in A. \text{subseq } y \ x\}$

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

shows $SUPSEQ \ \{\} = \{\}$
and $SUPSEQ \ \{\ \} = UNIV$

unfolding *SUPSEQ-def* **by** *auto*

lemma *SUPSEQ-atom* [*simp*]:

shows $SUPSEQ \ \{[c]\} = UNIV \cdot \{[c]\} \cdot UNIV$

unfolding *SUPSEQ-def conc-def*

by (*auto dest: list-emb-ConsD*)

lemma *SUPSEQ-union* [*simp*]:

shows $SUPSEQ \ (A \cup B) = SUPSEQ \ A \cup SUPSEQ \ B$

unfolding *SUPSEQ-def* **by** *auto*

lemma *SUPSEQ-conc* [*simp*]:

shows $SUPSEQ \ (A \cdot B) = SUPSEQ \ A \cdot SUPSEQ \ B$

unfolding *SUPSEQ-def conc-def*

apply (*auto*)

apply (*drule list-emb-appendD*)

apply(*auto*)
by (*metis list-emb-append-mono*)

lemma *SUPSEQ-star* [*simp*]:
shows $SUPSEQ (A^*) = UNIV$
apply(*subst star-unfold-left*)
apply(*simp only: SUPSEQ-union*)
apply(*simp*)
done

6.2 Regular expression that recognises every character

definition

$Allreg :: 'a::finite\ rexp$

where

$Allreg \equiv \biguplus (Atom\ 'a\ UNIV)$

lemma *Allreg-lang* [*simp*]:
shows $lang\ Allreg = (\bigcup a. \{[a]\})$
unfolding *Allreg-def* **by** *auto*

lemma [*simp*]:
shows $(\bigcup a. \{[a]\})^* = UNIV$
apply(*auto*)
apply(*induct-tac x*)
apply(*auto*)
apply(*subgoal-tac [a] @ list \in (\bigcup a. \{[a]\})^**)
apply(*simp*)
apply(*rule append-in-starI*)
apply(*auto*)
done

lemma *Star-Allreg-lang* [*simp*]:
shows $lang (Star\ Allreg) = UNIV$
by *simp*

fun

$UP :: 'a::finite\ rexp \Rightarrow 'a\ rexp$

where

$UP (Zero) = Zero$

| $UP (One) = Star\ Allreg$

| $UP (Atom\ c) = Times (Star\ Allreg) (Times (Atom\ c) (Star\ Allreg))$

| $UP (Plus\ r1\ r2) = Plus (UP\ r1) (UP\ r2)$

| $UP (Times\ r1\ r2) = Times (UP\ r1) (UP\ r2)$

| $UP (Star\ r) = Star\ Allreg$

lemma *lang-UP*:

fixes $r::'a::finite\ rexp$

shows $lang (UP\ r) = SUPSEQ (lang\ r)$

by (induct r) (simp-all)

lemma SUPSEQ-regular:

fixes $A::'a::\text{finite lang}$

assumes regular A

shows regular (SUPSEQ A)

proof –

from *assms* obtain $r::'a::\text{finite rexp}$ where $\text{lang } r = A$ by *auto*

then have $\text{lang } (UP\ r) = SUPSEQ\ A$ by (simp add: lang-UP)

then show regular (SUPSEQ A) by *auto*

qed

lemma SUPSEQ-subset:

fixes $A::'a::\text{finite list set}$

shows $A \subseteq SUPSEQ\ A$

unfolding SUPSEQ-def by *auto*

lemma SUBSEQ-complement:

shows $\neg (SUBSEQ\ A) = SUPSEQ\ (\neg (SUBSEQ\ A))$

proof –

have $\neg (SUBSEQ\ A) \subseteq SUPSEQ\ (\neg (SUBSEQ\ A))$

by (rule SUPSEQ-subset)

moreover

have $SUPSEQ\ (\neg (SUBSEQ\ A)) \subseteq \neg (SUBSEQ\ A)$

proof (rule ccontr)

assume $\neg (SUPSEQ\ (\neg (SUBSEQ\ A)) \subseteq \neg (SUBSEQ\ A))$

then obtain x where

$a: x \in SUPSEQ\ (\neg (SUBSEQ\ A))$ and

$b: x \notin \neg (SUBSEQ\ A)$ by *auto*

from a obtain y where $c: y \in \neg (SUBSEQ\ A)$ and $d: \text{subseq } y\ x$

by (auto simp add: SUPSEQ-def)

from b have $x \in SUBSEQ\ A$ by *simp*

then obtain x' where $f: x' \in A$ and $e: \text{subseq } x\ x'$

by (auto simp add: SUBSEQ-def)

from $d\ e$ have $\text{subseq } y\ x'$

by (rule subseq-order.order-trans)

then have $y \in SUBSEQ\ A$ using f

by (auto simp add: SUBSEQ-def)

with c show *False* by *simp*

qed

ultimately show $\neg (SUBSEQ\ A) = SUPSEQ\ (\neg (SUBSEQ\ A))$ by *simp*

qed

definition

$\text{minimal} :: 'a::\text{finite list} \Rightarrow 'a\ \text{lang} \Rightarrow \text{bool}$

where

$minimal\ x\ A \equiv (\forall y \in A. subseq\ y\ x \longrightarrow subseq\ x\ y)$

lemma *main-lemma*:

shows $\exists M. finite\ M \wedge SUPSEQ\ A = SUPSEQ\ M$

proof –

define M **where** $M = \{x \in A. minimal\ x\ A\}$

have *finite* M

unfolding *M-def* *minimal-def*

by (*rule* *subseq-Higman-antichains*) (*auto simp add: subseq-order.antisym*)

moreover

have $SUPSEQ\ A \subseteq SUPSEQ\ M$

proof

fix x

assume $x \in SUPSEQ\ A$

then obtain y **where** $y \in A$ **and** $subseq\ y\ x$ **by** (*auto simp add: SUPSEQ-def*)

then have $a: y \in \{y' \in A. subseq\ y'\ x\}$ **by** *simp*

obtain z **where** $b: z \in A\ subseq\ z\ x$ **and** $c: \forall y. subseq\ y\ z \wedge y \neq z \longrightarrow y \notin \{y' \in A. subseq\ y'\ x\}$

using *wfE-min[OF subseq-wf a]* **by** *auto*

then have $z \in M$

unfolding *M-def* *minimal-def*

by (*auto intro: subseq-order.order-trans*)

with $b(2)$ **show** $x \in SUPSEQ\ M$

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

qed

moreover

have $SUPSEQ\ M \subseteq SUPSEQ\ A$

by (*auto simp add: SUPSEQ-def M-def*)

ultimately

show $\exists M. finite\ M \wedge SUPSEQ\ A = SUPSEQ\ M$ **by** *blast*

qed

6.3 Closure of *SUBSEQ* and *SUPSEQ*

lemma *closure-SUPSEQ*:

fixes $A::'a::finite\ lang$

shows *regular* ($SUPSEQ\ A$)

proof –

obtain M **where** $a: finite\ M$ **and** $b: SUPSEQ\ A = SUPSEQ\ M$

using *main-lemma* **by** *blast*

have *regular* M **using** a **by** (*rule* *finite-regular*)

then have *regular* ($SUPSEQ\ M$) **by** (*rule* *SUPSEQ-regular*)

then show *regular* ($SUPSEQ\ A$) **using** b **by** *simp*

qed

lemma *closure-SUBSEQ*:

fixes $A::'a::finite\ lang$

shows *regular* ($SUBSEQ\ A$)

proof –

```

have regular (SUPSEQ (– SUBSEQ A)) by (rule closure-SUPSEQ)
then have regular (– SUBSEQ A) by (subst SUBSEQ-complement) (simp)
then have regular (– (– (SUBSEQ A))) by (rule closure-complement)
then show regular (SUBSEQ A) by simp
qed

end

```

7 Tools for showing non-regularity of a language

```

theory Non-Regular-Languages
  imports Myhill
begin

```

7.1 Auxiliary material

```

lemma bij-betw-image-quotient:
  bij-betw ( $\lambda y. f - \{y\}$ ) ( $f \text{ ' } A$ ) ( $A // \{(a,b). f a = f b\}$ )
  by (force simp: bij-betw-def inj-on-def image-image quotient-def)

```

```

lemma regular-Derivs-finite:
  fixes  $r :: 'a :: \text{finite rexp}$ 
  shows finite (range ( $\lambda w. \text{Derivs } w \text{ (lang } r)$ ))
proof –
  have  $?thesis \longleftrightarrow \text{finite} \text{ (UNIV // } \approx \text{lang } r)$ 
  unfolding str-eq-conv-Derivs by (rule bij-betw-finite bij-betw-image-quotient)+
  also have ... by (subst Myhill-Nerode [symmetric]) auto
  finally show  $?thesis$  .
qed

```

```

lemma Nil-in-Derivs-iff:  $\square \in \text{Derivs } w \text{ } A \longleftrightarrow w \in A$ 
  by (auto simp: Derivs-def)

```

The following operation repeats a list n times (usually written as w^n).

```

primrec repeat ::  $\text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  where
  repeat 0  $xs = []$ 
| repeat (Suc  $n$ )  $xs = xs @ \text{repeat } n \text{ } xs$ 

```

```

lemma repeat-Cons-left:  $\text{repeat} \text{ (Suc } n) \text{ } xs = xs @ \text{repeat } n \text{ } xs$  by simp

```

```

lemma repeat-Cons-right:  $\text{repeat} \text{ (Suc } n) \text{ } xs = \text{repeat } n \text{ } xs @ xs$ 
  by (induction n) simp-all

```

```

lemma repeat-Cons-append-commute [simp]:  $\text{repeat } n \text{ } xs @ xs = xs @ \text{repeat } n \text{ } xs$ 
  by (subst repeat-Cons-right [symmetric]) simp

```

```

lemma repeat-Cons-add [simp]:  $\text{repeat} \text{ (} m + n \text{) } xs = \text{repeat } m \text{ } xs @ \text{repeat } n \text{ } xs$ 
  by (induction m) simp-all

```

lemma *repeat-Nil* [*simp*]: $\text{repeat } n \ [] = []$
by (*induction n*) *simp-all*

lemma *repeat-conv-replicate*: $\text{repeat } n \ xs = \text{concat } (\text{replicate } n \ xs)$
by (*induction n*) *simp-all*

lemma *nth-prefixes* [*simp*]: $n \leq \text{length } xs \implies \text{prefixes } xs \ ! \ n = \text{take } n \ xs$
by (*induction xs arbitrary: n*) (*auto simp: nth-Cons split: nat.splits*)

lemma *nth-suffixes* [*simp*]: $n \leq \text{length } xs \implies \text{suffixes } xs \ ! \ n = \text{drop } (\text{length } xs - n) \ xs$
by (*subst suffixes-conv-prefixes*) (*simp-all add: rev-take*)

lemma *length-take-prefixes*:
assumes $xs \in \text{set } (\text{take } n \ (\text{prefixes } ys))$
shows $\text{length } xs < n$
proof (*cases n ≤ Suc (length ys)*)
case True
with *assms* **obtain** i **where** $i < n \ xs = \text{take } i \ ys$
by (*subst (asm) nth-image [symmetric]*) *auto*
thus *?thesis* **by** *simp*
next
case False
with *assms* **have** $\text{prefix } xs \ ys$ **by** *simp*
hence $\text{length } xs \leq \text{length } ys$ **by** (*rule prefix-length-le*)
also from *False* **have** $\dots < n$ **by** *simp*
finally show *?thesis* .
qed

7.2 Non-regularity by giving an infinite set of equivalence classes

Non-regularity can be shown by giving an infinite set of non-equivalent words (w.r.t. the Myhill–Nerode relation).

lemma *not-regular-langI*:
assumes $\text{infinite } B \wedge x \ y. \ x \in B \implies y \in B \implies x \neq y \implies \exists w. \neg(x @ w \in A \longleftrightarrow y @ w \in A)$
shows $\neg \text{regular-lang } (A :: 'a :: \text{finite list set})$
proof –
have $(\lambda w. \text{Derivs } w \ A) \ ' B \subseteq \text{range } (\lambda w. \text{Derivs } w \ A)$ **by** *blast*
moreover from *assms(2)* **have** $\text{inj-on } (\lambda w. \text{Derivs } w \ A) \ B$
by (*auto simp: inj-on-def Derivs-def*)
with *assms(1)* **have** $\text{infinite } ((\lambda w. \text{Derivs } w \ A) \ ' B)$
by (*blast dest: finite-imageD*)
ultimately have $\text{infinite } (\text{range } (\lambda w. \text{Derivs } w \ A))$ **by** (*rule infinite-super*)
with *regular-Derivs-finite* **show** *?thesis* **by** *blast*
qed

lemma *not-regular-langI'*:
assumes *infinite B* $\wedge x y. x \in B \implies y \in B \implies x \neq y \implies \exists w. \neg(f x @ w \in A \longleftrightarrow f y @ w \in A)$
shows \neg *regular-lang* (*A* :: 'a :: finite list set)
proof (*rule not-regular-langI*)
from *assms*(2) **have** *inj-on f B* **by** (*force simp: inj-on-def*)
with \langle *infinite B* \rangle **show** *infinite (f ' B)* **by** (*simp add: finite-image-iff*)
qed (*insert assms, auto*)

7.3 The Pumping Lemma

The Pumping lemma can be shown very easily from the Myhill–Nerode theorem: if we have a word whose length is more than the (finite) number of equivalence classes, then it must have two different prefixes in the same class and the difference between these two prefixes can then be “pumped”.

lemma *pumping-lemma-aux*:
fixes *A* :: 'a list set
defines $\delta \equiv \lambda w. \text{Derivs } w \ A$
defines $n \equiv \text{card } (\text{range } \delta)$
assumes $z \in A$ *finite (range δ)* *length z \geq n*
shows $\exists u v w. z = u @ v @ w \wedge \text{length } (u @ v) \leq n \wedge v \neq [] \wedge (\forall i. u @ \text{repeat } i \ v @ w \in A)$
proof –
define *P* **where** $P = \text{set } (\text{take } (\text{Suc } n) (\text{prefixes } z))$
from \langle *length z \geq n* \rangle **have** [*simp*]: $\text{card } P = \text{Suc } n$
unfolding *P-def* **by** (*subst distinct-card*) (*auto intro!: distinct-take*)
have *length-le*: *length y \leq n* **if** $y \in P$ **for** *y*
using *length-take-prefixes[OF that [unfolded P-def]]* **by** *simp*

have $\text{card } (\delta ' P) \leq \text{card } (\text{range } \delta)$ **by** (*intro card-mono assms*) *auto*
also from *assms* **have** $\dots < \text{card } P$ **by** *simp*
finally have \neg *inj-on δ P* **by** (*rule pigeonhole*)
then obtain *a b* **where** $ab: a \in P \ b \in P \ a \neq b \ \text{Derivs } a \ A = \text{Derivs } b \ A$
by (*auto simp: inj-on-def δ -def*)
from *ab* **have** *prefix-ab*: *prefix a z prefix b z* **by** (*auto simp: P-def dest: in-set-takeD*)
from *ab* **have** *length-ab*: *length a \leq n* *length b \leq n*
by (*simp-all add: length-le*)

have *: *?thesis*
if uz' : *prefix u z' prefix z' z* *length z' \leq n*
 $u \neq z' \ \text{Derivs } z' \ A = \text{Derivs } u \ A$ **for** $u \ z'$

proof –
from \langle *prefix u z'* \rangle **and** \langle $u \neq z'$ \rangle
obtain *v* **where** v [*simp*]: $z' = u @ v \ v \neq []$
by (*auto simp: prefix-def*)
from \langle *prefix z' z* \rangle **obtain** *w* **where** [*simp*]: $z = u @ v @ w$
by (*auto simp: prefix-def*)

hence [simp]: *Derivs* (repeat i v) (*Derivs* u A) = *Derivs* u A **for** i
by (induction i) (use uz' **in** *simp-all*)
have *Derivs* z A = *Derivs* (u @ repeat i v @ w) A **for** i
using uz' **by** *simp*
with $\langle z \in A \rangle$ **and** uz' **have** $\forall i. u$ @ repeat i v @ $w \in A$
by (*simp add: Nil-in-Derivs-iff* [of - A , symmetric])
moreover **have** $z = u$ @ v @ w **by** *simp*
moreover **from** $\langle \text{length } z' \leq n \rangle$ **have** $\text{length } (u$ @ $v) \leq n$ **by** *simp*
ultimately **show** ?thesis **using** $\langle v \neq [] \rangle$ **by** *blast*
qed

from *prefix-ab* **have** *prefix* a $b \vee$ *prefix* b a **by** (rule *prefix-same-cases*)
with *[of a b] **and** *[of b a] **and** *ab* **and** *prefix-ab* **and** *length-ab* **show** ?thesis
by *blast*
qed

theorem *pumping-lemma*:

fixes $r :: 'a :: \text{finite rexp}$

obtains n **where**

$\bigwedge z. z \in \text{lang } r \implies \text{length } z \geq n \implies$

$\exists u v w. z = u$ @ v @ $w \wedge \text{length } (u$ @ $v) \leq n \wedge v \neq [] \wedge (\forall i. u$ @ repeat i v @ $w \in \text{lang } r)$

proof –

let $?n = \text{card } (\text{range } (\lambda w. \text{Derivs } w (\text{lang } r)))$

have $\exists u v w. z = u$ @ v @ $w \wedge \text{length } (u$ @ $v) \leq ?n \wedge v \neq [] \wedge (\forall i. u$ @ repeat i v @ $w \in \text{lang } r)$

if $z \in \text{lang } r$ **and** $\text{length } z \geq ?n$ **for** z

by (*intro pumping-lemma-aux*[of z] that *regular-Derivs-finite*)

thus ?thesis **by** (rule that)

qed

corollary *pumping-lemma-not-regular-lang*:

fixes $A :: 'a :: \text{finite list set}$

assumes $\bigwedge n. \text{length } (z n) \geq n$ **and** $\bigwedge n. z n \in A$

assumes $\bigwedge n u v w. z n = u$ @ v @ $w \implies \text{length } (u$ @ $v) \leq n \implies v \neq [] \implies$
 u @ repeat (i n u v w) v @ $w \notin A$

shows $\neg \text{regular-lang } A$

proof

assume *regular-lang* A

then **obtain** r **where** $\text{lang } r = A$ **by** *blast*

from *pumping-lemma*[of r] **obtain** n

where $z n \in \text{lang } r \implies n \leq \text{length } (z n) \implies$

$\exists u v w. z n = u$ @ v @ $w \wedge \text{length } (u$ @ $v) \leq n \wedge v \neq [] \wedge (\forall i. u$ @ repeat i v @ $w \in \text{lang } r)$

by *metis*

from this **and** *assms*[of n] **obtain** $u v w$

where $z n = u$ @ v @ w **and** $\text{length } (u$ @ $v) \leq n$ **and** $v \neq []$ **and**

$\bigwedge i. u$ @ repeat i v @ $w \in \text{lang } r$ **by** (*auto simp: r*)

with *assms*(3)[of n u v w] **show** *False* **by** (*auto simp: r*)

qed

7.4 Examples

The language of all words containing the same number of *as* and *bs* is not regular.

lemma \neg regular-lang $\{w. \text{length } (\text{filter id } w) = \text{length } (\text{filter Not } w)\}$ (is \neg regular-lang ?A)

proof (rule not-regular-langI')

show infinite (UNIV :: nat set) by simp

next

fix $m n :: \text{nat}$ assume $m \neq n$

hence replicate m True @ replicate m False \in ?A and

replicate n True @ replicate m False \notin ?A by simp-all

thus $\exists w. \neg(\text{replicate } m \text{ True } @ w \in ?A \longleftrightarrow \text{replicate } n \text{ True } @ w \in ?A)$ by blast

qed

The language $\{a^i b^i \mid i \in \mathbb{N}\}$ is not regular.

lemma eq-replicate-iff:

$xs = \text{replicate } n \ x \longleftrightarrow \text{set } xs \subseteq \{x\} \wedge \text{length } xs = n$

using replicate-length-same[of $xs \ x$] by (subst eq-commute) auto

lemma replicate-eq-appendE:

assumes $xs @ ys = \text{replicate } n \ x$

obtains $i \ j$ where $n = i + j$ $xs = \text{replicate } i \ x$ $ys = \text{replicate } j \ x$

proof –

have $n = \text{length } (\text{replicate } n \ x)$ by simp

also note *assms* [symmetric]

finally have $n = \text{length } xs + \text{length } ys$ by simp

moreover have $xs = \text{replicate } (\text{length } xs) \ x$ and $ys = \text{replicate } (\text{length } ys) \ x$

using *assms* by (auto simp: eq-replicate-iff)

ultimately show ?thesis using that[of $\text{length } xs \ \text{length } ys$] by auto

qed

lemma \neg regular-lang (range $(\lambda i. \text{replicate } i \ \text{True } @ \text{replicate } i \ \text{False})$) (is \neg regular-lang ?A)

proof (rule pumping-lemma-not-regular-lang)

fix $n :: \text{nat}$

show $\text{length } (\text{replicate } n \ \text{True } @ \text{replicate } n \ \text{False}) \geq n$ by simp

show $\text{replicate } n \ \text{True } @ \text{replicate } n \ \text{False} \in ?A$ by simp

next

fix $n :: \text{nat}$ and $u \ v \ w :: \text{bool list}$

assume *decomp*: $\text{replicate } n \ \text{True } @ \text{replicate } n \ \text{False} = u @ v @ w$

and *length-le*: $\text{length } (u @ v) \leq n$ and *v-ne*: $v \neq []$

define $w1 \ w2$ where $w1 = \text{take } (n - \text{length } (u @ v)) \ w$ and $w2 = \text{drop } (n - \text{length } (u @ v)) \ w$

have *w-decomp*: $w = w1 @ w2$ by (simp add: *w1-def w2-def*)

```

have replicate n True = take n (replicate n True @ replicate n False) by simp
also note decomp
also have take n (u @ v @ w) = u @ v @ w1 using length-le by (simp add:
w1-def)
finally have u @ v @ w1 = replicate n True by simp
then obtain i j k
  where uvw1: n = i + j + k u = replicate i True v = replicate j True w1 =
replicate k True
  by (elim replicate-eq-appendE) auto

have replicate n False = drop n (replicate n True @ replicate n False) by simp
also note decomp
finally have [simp]: w2 = replicate n False using length-le by (simp add: w2-def)

have u @ repeat (Suc (Suc 0)) v @ w = replicate (n + j) True @ replicate n
False
  by (simp add: uvw1 w-decomp replicate-add [symmetric])
also have ...  $\notin$  ?A
proof safe
  fix m assume *: replicate (n + j) True @ replicate n False =
      replicate m True @ replicate m False (is ?lhs = ?rhs)
  have n = length (filter Not ?lhs) by simp
  also note *
  also have length (filter Not ?rhs) = m by simp
  finally have [simp]: m = n by simp
  from * have v = [] by (simp add: uvw1)
  with  $\langle v \neq [] \rangle$  show False by contradiction
qed
finally show u @ repeat (Suc (Suc 0)) v @ w  $\notin$  ?A .
qed

end

```

References

- [1] V. Antimirov. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science*, 155:291–319, 1995.
- [2] C. Wu, X. Zhang, and C. Urban. A Formalisation of the Myhill-Nerode Theorem based on Regular Expressions (Proof Pearl). In *Proc. of the 2nd International Conference on Interactive Theorem Proving*, volume 6898 of *LNCS*, pages 341–356, 2011.