The Myhill-Nerode Theorem Based on Regular Expressions

Chunhan Wu, Xingyuan Zhang and Christian Urban

October 13, 2025

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

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

Plus-combination for a set of regular expressions

abbreviation
  $Setalt :: 'a\ rexp\ set \Rightarrow 'a\ rexp\ (\langle \biguplus \text{-} \rangle\ [1000]\ 999)$
where
  $\biguplus A \equiv folds\ Plus\ Zero\ A$

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

lemma *folds-plus-simp* [*simp*]:
  fixes *rs*::(*'a rexp*) *set*
  assumes *a*: *finite rs*
  shows *lang* ($\biguplus$ *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* ⇒ *regular language*

notation
  conc (**infixr** ‹·› *100*) **and**
  star (‹-⋆› [*101*] *102*)

lemma *Pair-Collect* [*simp*]:
  shows (*x*, *y*) ∈ {(*x*, *y*). *P x y*} ⟷ *P x y*
by *simp*

Myhill-Nerode relation

definition
  str-eq :: *'a lang* ⇒ (*'a list* × *'a list*) *set* (‹≈-› [*100*] *100*)
where
  ≈*A* ≡ {(*x*, *y*). (∀ *z*. *x* @ *z* ∈ *A* ⟷ *y* @ *z* ∈ *A*)}

abbreviation
  str-eq-applied :: *'a list* ⇒ *'a lang* ⇒ *'a list* ⇒ *bool* (‹- ≈- -›)
where
  x ≈*A* *y* ≡ (*x*, *y*) ∈ ≈*A*

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

definition
  finals :: *'a lang* ⇒ *'a lang set*

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

lemma *lang-is-union-of-finals*:
  shows $A = \bigcup(\textit{finals } A)$
unfolding *finals-def*
unfolding *Image-def*
unfolding *str-eq-def*
by (*auto*) (*metis append-Nil2*)

lemma *finals-in-partitions*:
  shows *finals* $A \subseteq (\textit{UNIV} \ // \approx A)$
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* ⇒ *'a lang*
where
  lang-trm (*Lam r*) = *lang r*
| *lang-trm* (*Trn X r*) = $X \cdot$ *lang r*

fun
  lang-rhs::(*'a trm*) *set* ⇒ *'a lang*
where
  lang-rhs rhs = $\bigcup$ (*lang-trm* ` *rhs*)

lemma *lang-rhs-set*:
  shows *lang-rhs* {*Trn X r* | *r*. *P r*} = $\bigcup${*lang-trm* (*Trn X r*) | *r*. *P r*}
by (*auto*)

lemma *lang-rhs-union-distrib*:
  shows *lang-rhs A* ∪ *lang-rhs B* = *lang-rhs* (*A* ∪ *B*)
by *simp*

Transitions between equivalence classes

definition
  transition :: *'a lang* ⇒ *'a* ⇒ *'a lang* ⇒ *bool* (‹- ⊨-⇒-› [*100*,*100*,*100*] *100*)
where
  $Y \models c \Rightarrow X \equiv Y \cdot \{[c]\} \subseteq X$

Initial equational system

definition

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

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

2.2 Arden Operation on equations

fun
$Append\text{-}rexp :: 'a\ rexp \Rightarrow 'a\ trm \Rightarrow 'a\ trm$
where
$Append\text{-}rexp\ r\ (Lam\ rexp) \ \ = Lam\ (Times\ rexp\ r)$
$\mid Append\text{-}rexp\ r\ (Trn\ X\ rexp) = Trn\ X\ (Times\ rexp\ r)$

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

definition
$Arden\ X\ rhs \equiv$
$\quad Append\text{-}rexp\text{-}rhs\ (rhs - \{Trn\ X\ r \mid r.\ Trn\ X\ r \in rhs\})\ (Star\ (\biguplus\ \{r.\ Trn\ X\ r \in rhs\}))$

2.3 Substitution Operation on equations

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

definition
$Subst\text{-}all :: ('a\ lang \times ('a\ trm)\ set)\ set \Rightarrow 'a\ lang \Rightarrow ('a\ trm)\ set \Rightarrow ('a\ lang \times ('a\ trm)\ set)\ set$
where
$Subst\text{-}all\ ES\ X\ xrhs \equiv \{(Y,\ Subst\ yrhs\ X\ xrhs) \mid Y\ yrhs.\ (Y,\ yrhs) \in ES\}$

definition
$Remove\ ES\ X\ xrhs \equiv$
$\quad Subst\text{-}all\ \ (ES - \{(X,\ xrhs)\})\ X\ (Arden\ X\ xrhs)$

2.4 While-combinator and invariants

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

lemma *IterI2*:

  assumes $(Y,\ yrhs) \in ES$
  and $X \neq Y$
  and $\bigwedge Y\ yrhs.\ [\![(Y,\ yrhs) \in ES;\ X \neq Y]\!] \Longrightarrow Q\ (Remove\ ES\ Y\ yrhs)$
  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
  $soundness\ ES \equiv \forall (X,\ rhs) \in ES.\ X = lang\text{-}rhs\ rhs$

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

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

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

lemma *finite-rhs-def2*:
  $finite\text{-}rhs\ ES = (\forall\ X\ rhs.\ (X,\ rhs) \in ES \longrightarrow finite\ rhs)$
unfolding *finite-rhs-def* **by** *auto*

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

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

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

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

lemma *lhss-union-distrib*:
  shows $lhss\ (A \cup B) = lhss\ A \cup lhss\ B$

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

definition
invariant ES ≡ *finite ES*
∧ *finite-rhs ES*
∧ *soundness ES*
∧ *distinctness ES*
∧ *ardenable-all ES*
∧ *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 [[*simproc add*: *finite-Collect*]]

lemma *finite-Trn*:
assumes *fin*: *finite rhs*
shows *finite* {*r*. *Trn Y r* ∈ *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* ∈ *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*| *r*. *Trn X r* ∈ *rhs*}) = *X* · (*lang* (⨄{*r*. *Trn X r* ∈ *rhs*}))
proof −
have *finite* {*r*. *Trn X r* ∈ *rhs*}
by (*rule finite-Trn*[*OF finite*])
then show *lang-rhs* ({*Trn X r*| *r*. *Trn X r* ∈ *rhs*}) = *X* · (*lang* (⨄{*r*. *Trn X r* ∈ *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* · *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 · 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 ∈ X X ∈ UNIV // ≈A
  shows X = ≈A `` {s}
using assms
unfolding quotient-def Image-def str-eq-def
by auto

lemma every-eqclass-has-transition:
  assumes has-str: s @ [c] ∈ X
  and     in-CS:   X ∈ UNIV // ≈A
  obtains Y where Y ∈ UNIV // ≈A and Y · {[c]} ⊆ X and s ∈ Y
proof −
  define Y where Y = ≈A `` {s}
  have Y ∈ UNIV // ≈A
    unfolding Y-def quotient-def by auto
  moreover
  have X = ≈A `` {s @ [c]}
    using has-str in-CS defined-by-str by blast
  then have Y · {[c]} ⊆ X
    unfolding Y-def Image-def conc-def
    unfolding str-eq-def
    by clarsimp
  moreover
  have s ∈ 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) ∈ Init (UNIV // ≈A)
  shows X = lang-rhs rhs
proof
  show X ⊆ lang-rhs rhs
  proof
    fix x
    assume in-X: x ∈ X
    { assume empty: x = []
      then have x ∈ lang-rhs rhs using X-in-eqs in-X
        unfolding Init-def Init-rhs-def
        by auto
    }
    moreover
    { assume not-empty: x ≠ []
```

```
      then obtain s c where decom: x = s @ [c]
        using rev-cases by blast
      have X ∈ UNIV // ≈A using X-in-eqs unfolding Init-def by auto
      then obtain Y where Y ∈ UNIV // ≈A Y · {[c]} ⊆ X s ∈ Y
        using decom in-X every-eqclass-has-transition by metis
     then have x ∈ lang-rhs {Trn Y (Atom c)| Y c. Y ∈ UNIV // ≈A ∧ Y ⊨c⇒
X}
        unfolding transition-def
        using decom by (fastforce simp add: conc-def)
      then have x ∈ lang-rhs rhs using X-in-eqs in-X
        unfolding Init-def Init-rhs-def by simp
    }
    ultimately show x ∈ lang-rhs rhs by blast
  qed
next
  show lang-rhs rhs ⊆ 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 // ≈A)
  shows invariant (Init (UNIV // ≈A))
proof (rule invariantI)
  show soundness (Init (UNIV // ≈A))
    unfolding soundness-def
    using l-eq-r-in-eqs by auto
  show finite (Init (UNIV // ≈A)) using finite-CS
    unfolding Init-def by simp
  show distinctness (Init (UNIV // ≈A))
    unfolding distinctness-def Init-def by simp
  show ardenable-all (Init (UNIV // ≈A))
    unfolding ardenable-all-def Init-def Init-rhs-def ardenable-def
   by auto
  show finite-rhs (Init (UNIV // ≈A))
    using finite-Init-rhs[OF finite-CS]
    unfolding finite-rhs-def Init-def by auto
  show validity (Init (UNIV // ≈A))
    unfolding validity-def Init-def Init-rhs-def rhss-def lhss-def
    by auto
```

```
qed
```

2.6 Interations

```
lemma Arden-preserves-soundness:
  assumes l-eq-r: X = lang-rhs rhs
  and not-empty: ardenable rhs
  and finite: finite rhs
  shows X = lang-rhs (Arden X rhs)
proof −
  define A where A = lang (⨄{r. Trn X r ∈ rhs})
  define b where b = {Trn X r | r. Trn X r ∈ rhs}
  define B where B = lang-rhs (rhs − b)
  have not-empty2: [] ∉ A
    using finite-Trn[OF finite] not-empty
    unfolding A-def ardenable-def by simp
  have X = lang-rhs rhs using l-eq-r by simp
  also have ... = lang-rhs (b ∪ (rhs − b)) unfolding b-def by auto
  also have ... = lang-rhs b ∪ B unfolding B-def by (simp only: lang-rhs-union-distrib)
  also have ... = X · A ∪ B
    unfolding b-def
    unfolding trm-soundness[OF finite]
    unfolding A-def
    by blast
  finally have X = X · A ∪ B .
  then have X = B · A⋆
    by (simp add: reversed-Arden[OF not-empty2])
  also have ... = lang-rhs (Arden X rhs)
    unfolding Arden-def A-def B-def b-def
    by (simp only: lang-of-append-rexp-rhs lang.simps)
  finally show X = lang-rhs (Arden X rhs) by simp
qed

lemma Append-preserves-finite:
  finite rhs ⟹ finite (Append-rexp-rhs rhs r)
by (auto simp: Append-rexp-rhs-def)

lemma Arden-preserves-finite:
  finite rhs ⟹ finite (Arden X rhs)
by (auto simp: Arden-def Append-preserves-finite)

lemma Append-preserves-ardenable:
  ardenable rhs ⟹ 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 ⟹ ardenable (rhs − A)
by (auto simp:ardenable-def)
```

lemma *ardenable-set-union*:
  $[\![$*ardenable rhs*; *ardenable rhs'*$]\!] \Longrightarrow$ *ardenable* (*rhs* $\cup$ *rhs'*)
by (*auto simp*:*ardenable-def*)

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

lemma *Subst-preserves-ardenable*:
  $[\![$*ardenable rhs*; *ardenable xrhs*$]\!] \Longrightarrow$ *ardenable* (*Subst rhs X xrhs*)
by (*simp only*: *Subst-def Append-preserves-ardenable ardenable-set-union ardenable-set-sub*)

lemma *Subst-preserves-soundness*:
  assumes *substor*: *X* = *lang-rhs xrhs*
  and *finite*: *finite rhs*
  shows *lang-rhs* (*Subst rhs X xrhs*) = *lang-rhs rhs* (**is** *?Left* = *?Right*)
proof−
  define *A* **where** *A* = *lang-rhs* (*rhs* − {*Trn X r* | *r*. *Trn X r* ∈ *rhs*})
  have *?Left* = *A* ∪ *lang-rhs* (*Append-rexp-rhs xrhs* ($\biguplus$ {*r*. *Trn X r* ∈ *rhs*}))
    unfolding *Subst-def*
    unfolding *lang-rhs-union-distrib*[*symmetric*]
    by (*simp add*: *A-def*)
  moreover have *?Right* = *A* ∪ *lang-rhs* {*Trn X r* | *r*. *Trn X r* ∈ *rhs*}
  proof−
    have *rhs* = (*rhs* − {*Trn X r* | *r*. *Trn X r* ∈ *rhs*}) ∪ ({*Trn X r* | *r*. *Trn X r* ∈ *rhs*}) **by** *auto*
    thus *?thesis*
      unfolding *A-def*
      unfolding *lang-rhs-union-distrib*
      by *simp*
  qed
  moreover
  have *lang-rhs* (*Append-rexp-rhs xrhs* ($\biguplus$ {*r*. *Trn X r* ∈ *rhs*})) = *lang-rhs* {*Trn X r* | *r*. *Trn X r* ∈ *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*:
  $[\![$*finite rhs*; *finite yrhs*$]\!] \Longrightarrow$ *finite* (*Subst rhs Y yrhs*)
by (*auto simp*: *Subst-def Append-preserves-finite*)

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

```
declare [[simproc del: finite-Collect]]

lemma Subst-all-preserves-finite-rhs:
  ⟦finite-rhs ES; finite yrhs⟧ ⟹ 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:
  rhss (Append-rexp-rhs rhs r) = 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:
  rhss (Arden Y yrhs) = rhss yrhs − {Y}
apply (simp add:Arden-def append-rhs-preserves-cls)
by (auto simp: rhss-def)

lemma lhss-preserves-cls:
  lhss (Subst-all ES Y yrhs) = lhss ES
by (auto simp: lhss-def Subst-all-def)

lemma Subst-updates-cls:
  X ∉ rhss xrhs ⟹
      rhss (Subst rhs X xrhs) = rhss rhs ∪ 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 ∪ {(Y, yrhs)})        (is validity ?A)
  shows validity (Subst-all ES Y (Arden Y yrhs))  (is validity ?B)
proof −
  { fix X xrhs'
    assume (X, xrhs') ∈ ?B
    then obtain xrhs
      where xrhs-xrhs': xrhs' = Subst xrhs Y (Arden Y yrhs)
      and X-in: (X, xrhs) ∈ ES by (simp add:Subst-all-def, blast)
    have rhss xrhs' ⊆ lhss ?B
    proof−
      have lhss ?B = lhss ES by (auto simp add:lhss-def Subst-all-def)
      moreover have rhss xrhs' ⊆ lhss ES
      proof−
        have rhss xrhs' ⊆  rhss xrhs ∪ rhss (Arden Y yrhs) − {Y}
        proof −
          have Y ∉ 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 ⊆ lhss ES ∪ {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) ⊆ lhss ES ∪ {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 ∪ {(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) ∈ 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 ∃(Y, yrhs) ∈ ES. (X, xrhs) ≠ (Y, yrhs)
proof−
  have card ES > 1 using card e-in finite
    by (cases card ES) (auto)
  then have card (ES − {(X, xrhs)}) > 0
    using finite e-in by auto
  then have (ES − {(X, xrhs)}) ≠ {} using finite by (rule-tac notI, simp)
  then show ∃(Y, yrhs) ∈ ES. (X, xrhs) ≠ (Y, yrhs)
    by auto
qed

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

lemma iteration-step-invariant:
  assumes Inv-ES: invariant ES
  and     X-in-ES: (X, xrhs) ∈ ES
  and     Cnd: Cond ES
  shows invariant (Iter X ES)
proof −
  have finite-ES: finite ES using Inv-ES by (simp add: invariant-def)
  then obtain Y yrhs
    where Y-in-ES: (Y, yrhs) ∈ ES and not-eq: (X, xrhs) ≠ (Y, yrhs)
    using Cnd X-in-ES by (drule-tac card-noteq-1-has-more) (auto)
  then have (Y, yrhs) ∈ ES X ≠ Y
    using X-in-ES Inv-ES unfolding invariant-def distinctness-def
    by auto
  then show invariant (Iter X ES)
  proof(rule IterI2)
    fix Y yrhs
    assume h: (Y, yrhs) ∈ ES X ≠ Y
    then have ES − {(Y, yrhs)} ∪ {(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) ∈ ES
  and     Cnd: Cond ES
  shows ∃xrhs'. (X, xrhs') ∈ (Iter X ES)
proof −
  have finite-ES: finite ES using Inv-ES by (simp add: invariant-def)
  then obtain Y yrhs
    where (Y, yrhs) ∈ ES (X, xrhs) ≠ (Y, yrhs)
    using Cnd X-in-ES by (drule-tac card-noteq-1-has-more) (auto)
  then have (Y, yrhs) ∈ ES  X ≠ Y
    using X-in-ES Inv-ES unfolding invariant-def distinctness-def
    by auto
  then show ∃xrhs'. (X, xrhs') ∈ (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 ∈ (UNIV // ≈A)
  shows ∃rhs. Solve X (Init (UNIV // ≈A)) = {(X, rhs)} ∧ invariant {(X, rhs)}
proof −
  define Inv where Inv ES ⟷ invariant ES ∧ (∃rhs. (X, rhs) ∈ 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: ¬Cond ES
    from inv obtain rhs where (X, rhs) ∈ ES unfolding Inv-def by auto
    moreover
    from not-crd have card ES = 1 by simp
    ultimately
    have ES = {(X, rhs)} by (auto simp add: card-Suc-eq)
    then have ∃ rhs'. ES = {(X, rhs')} ∧ 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 (Iter X ES, ES) ∈ measure card
      unfolding Inv-def
      apply(clarify)
      apply(rule-tac iteration-step-measure)
      apply(auto)
      done }
  ultimately
  show ∃ rhs. Solve X (Init (UNIV // ≈A)) = {(X, rhs)} ∧ invariant {(X, rhs)}
    unfolding Solve-def by (rule while-rule)
qed

lemma every-eqcl-has-reg:
  fixes A::('a::finite) lang
  assumes finite-CS: finite (UNIV // ≈A)
  and X-in-CS: X ∈ (UNIV // ≈A)
  shows ∃ r. X = 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 = Arden X xrhs
  have rhss xrhs ⊆ {X} using Inv-ES
    unfolding validity-def invariant-def rhss-def lhss-def
    by auto
  then have rhss A = {} unfolding A-def
    by (simp add: Arden-removes-cl)
  then have eq: {Lam r | r. Lam r ∈ 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 ∈ A} by (rule finite-Lam)

  have X = lang-rhs xrhs using Inv-ES unfolding invariant-def soundness-def
    by simp
  then have X = 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 = lang-rhs {Lam r | r. Lam r ∈ A} using eq by simp
  then have X = lang (⨄{r. Lam r ∈ A}) using fin by auto
  then show ∃ r. X = lang r by blast
qed

lemma bchoice-finite-set:
  assumes a: ∀ x ∈ S. ∃ y. x = f y
  and     b: finite S
  shows ∃ ys. (⋃ S) = ⋃(f ‘ ys) ∧ 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 // ≈A)
  shows   ∃ r. A = lang r
proof −
  have fin: finite (finals A)
    using finals-in-partitions finite-CS by (rule finite-subset)
  have ∀ X ∈ (UNIV // ≈A). ∃ r. X = lang r
    using finite-CS every-eqcl-has-reg by blast
  then have a: ∀ X ∈ finals A. ∃ r. X = lang r
    using finals-in-partitions by auto
  then obtain rs::('a rexp) set where ⋃ (finals A) = ⋃(lang ‘ rs) finite rs
    using fin by (auto dest: bchoice-finite-set)
  then have A = lang (⨄rs)
    unfolding lang-is-union-of-finals[symmetric] by simp
  then show ∃ r. A = lang r by blast
qed


end

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

3 Second direction of MN: *regular language ⇒ finite partition*

3.1 Tagging functions

```
definition
   tag-eq :: ('a list ⇒ 'b) ⇒ ('a list × 'a list) set (‹=-=›)
where
```

```
  =tag= ≡ {(x, y). tag x = tag y}

abbreviation
  tag-eq-applied :: 'a list ⇒ ('a list ⇒ 'b) ⇒ 'a list ⇒ bool (‹- =-= -›)
where
  x =tag= y ≡ (x, y) ∈ =tag=

lemma [simp]:
 shows (≈A) `` {x} = (≈A) `` {y} ⟷ x ≈A y
unfolding str-eq-def by auto

lemma refined-intro:
 assumes ⋀x y z. ⟦x =tag= y; x @ z ∈ A⟧ ⟹ y @ z ∈ A
 shows =tag= ⊆ ≈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 = λX. tag ` X and ?A = (UNIV // =tag=)
 have finite (?f ` ?A)
 proof –
  have range ?f ⊆ (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 ⊆ 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 ∈ ?A
    and  Y-in: Y ∈ ?A
    and  tag-eq: ?f X = ?f Y
   then obtain x y
    where x ∈ X y ∈ 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 ⊆ R2
  and eq1: equiv UNIV R1 and eq2: equiv UNIV R2
  shows finite (UNIV // R2)
proof –
  let ?f = λX. {R1 `` {x} | x. x ∈ X}
    and ?A = UNIV // R2 and ?B = UNIV // R1
  have ?f ` ?A ⊆ 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 ∈ ?A and Y-in: Y ∈ ?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 ∈ X by simp
      then have R1 ``{x} ∈ ?f X by auto
      with eq-f have R1 `` {x} ∈ ?f Y by simp
      then obtain y
        where y-in: y ∈ Y and eq-r1-xy: R1 `` {x} = R1 `` {y} by auto
      with eq-equiv-class[OF - eq1]
      have (x, y) ∈ R1 by blast
      with refined have (x, y) ∈ 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= ⊆ ≈A
  shows finite (UNIV // ≈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= ⊆ ≈A using refined .
next
  show equiv UNIV =tag=
  and  equiv UNIV (≈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 // ≈{} = {UNIV}
unfolding quotient-def Image-def str-eq-def by auto

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


lemma quot-one-subset:
  shows UNIV // ≈{[]} ⊆ {{[]}, UNIV − {[]}}
proof
  fix x
  assume x ∈ UNIV // ≈{[]}
  then obtain y where h: x = {z. y ≈{[]} z}
    unfolding quotient-def Image-def by blast
  { assume y = []
    with h have x = {[]} by (auto simp: str-eq-def)
    then have x ∈ {{[]}, UNIV − {[]}} by simp }
  moreover
  { assume y ≠ []
    with h have x = UNIV − {[]} by (auto simp: str-eq-def)
    then have x ∈ {{[]}, UNIV − {[]}} by simp }
  ultimately show x ∈ {{[]}, UNIV − {[]}} by blast
qed

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


lemma quot-atom-subset:
  UNIV // (≈{[c]}) ⊆ {{[]},{[c]}, UNIV − {[], [c]}}
proof
  fix x
  assume x ∈ UNIV // ≈{[c]}
  then obtain y where h: x = {z. (y, z) ∈ ≈{[c]}}
    unfolding quotient-def Image-def by blast
  show x ∈ {{[]},{[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 ≠ [] and y ≠ [c]
      hence ∀ z. (y @ z) ≠ [c] by (case-tac y, auto)
      moreover have ⋀ p. (p ≠ [] ∧ p ≠ [c]) = (∀ q. p @ q ≠ [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 // ≈{[c]})
by (rule finite-subset[OF quot-atom-subset]) (simp)
```

3.3 Case for *Plus*

```
definition
  tag-Plus :: 'a lang ⇒ 'a lang ⇒ 'a list ⇒ ('a lang × 'a lang)
where
  tag-Plus A B ≡ λx. (≈A `` {x}, ≈B `` {x})

lemma quot-plus-finiteI [intro]:
  assumes finite1: finite (UNIV // ≈A)
  and     finite2: finite (UNIV // ≈B)
  shows finite (UNIV // ≈(A ∪ B))
proof (rule-tac tag = tag-Plus A B in tag-finite-imageD)
  have finite ((UNIV // ≈A) × (UNIV // ≈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= ⊆ ≈(A ∪ B)
    unfolding tag-eq-def tag-Plus-def str-eq-def by auto
qed
```

3.4 Case for *Times*

```
definition
  Partitions x ≡ {(x_p, x_s). x_p @ x_s = x}

lemma conc-partitions-elim:
  assumes x ∈ A · B
  shows ∃ (u, v) ∈ Partitions x. u ∈ A ∧ v ∈ B
using assms unfolding conc-def Partitions-def
by auto

lemma conc-partitions-intro:
  assumes (u, v) ∈ Partitions x ∧ u ∈ A ∧  v ∈ B
  shows x ∈ A · B
using assms unfolding conc-def Partitions-def
by auto

lemma equiv-class-member:
  assumes x ∈ A
  and ≈A `` {x} = ≈A `` {y}
  shows y ∈ A
using assms
apply(simp)
apply(simp add: str-eq-def)
apply(metis append-Nil2)
done

definition
  tag-Times :: 'a lang ⇒ 'a lang ⇒ 'a list ⇒ 'a lang × 'a lang set
where
  tag-Times A B ≡ λx. (≈A `` {x}, {(≈B `` {x_s}) | x_p x_s. x_p ∈ A ∧ (x_p, x_s) ∈
Partitions x})

lemma tag-Times-injI:
  assumes a: tag-Times A B x = tag-Times A B y
  and     c: x @ z ∈ A · B
  shows y @ z ∈ A · B
proof -
  from c obtain u v where
    h1: (u, v) ∈ Partitions (x @ z) and
    h2: u ∈ A and
    h3: v ∈ 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 ∧ us @ z = v) ∨ (x @ us = u ∧ z = us @ v)
    by (auto simp add: append-eq-append-conv2)
  moreover
  { assume eq: x = u @ us us @ z = v
    have (≈B `` {us}) ∈ snd (tag-Times A B x)
      unfolding Partitions-def tag-Times-def using h2 eq
      by (auto simp add: str-eq-def)
    then have (≈B `` {us}) ∈ snd (tag-Times A B y)
      using a by simp
    then obtain u' us' where
      q1: u' ∈ A and
      q2: ≈B `` {us} = ≈B `` {us'} and
      q3: (u', us') ∈ Partitions y
      unfolding tag-Times-def by auto
    from q2 h3 eq
    have us' @ z ∈ B
      unfolding Image-def str-eq-def by auto
    then have y @ z ∈ A · B using q1 q3
      unfolding Partitions-def by auto
  }
  moreover
  { assume eq: x @ us = u z = us @ v
    have (≈A `` {x}) = fst (tag-Times A B x)
      by (simp add: tag-Times-def)
    then have (≈A `` {x}) = fst (tag-Times A B y)
      using a by simp
    then have ≈A `` {x} = ≈A `` {y}
      by (simp add: tag-Times-def)
    moreover
    have x @ us ∈ A using h2 eq by simp
    ultimately
    have y @ us ∈ A using equiv-class-member
      unfolding Image-def str-eq-def by blast
    then have (y @ us) @ v ∈ A · B
      using h3 unfolding conc-def by blast
    then have y @ z ∈ A · B using eq by simp
  }
  ultimately show y @ z ∈ A · B by blast
qed

lemma quot-conc-finiteI [intro]:
  assumes fin1: finite (UNIV // ≈A)
  and     fin2: finite (UNIV // ≈B)
  shows finite (UNIV // ≈(A · B))
proof (rule-tac tag = tag-Times A B in tag-finite-imageD)
  have ⋀x y z. ⟦tag-Times A B x = tag-Times A B y; x @ z ∈ A · B⟧ ⟹ y @ z
∈ A · B
    by (rule tag-Times-injI)
       (auto simp add: tag-Times-def tag-eq-def)
  then show =tag-Times A B= ⊆ ≈(A · B)
    by (rule refined-intro)
       (auto simp add: tag-eq-def)
next
  have *: finite ((UNIV // ≈A) × (Pow (UNIV // ≈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 $\exists (u, v) \in$ *Partitions* (x @ z). *strict-prefix* u $x \wedge u \in A\star \wedge v \in A\star$
proof –
  have ([], x @ z) $\in$ *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 $\exists (u, v) \in$ *Partitions* (x @ z). *strict-prefix* u $x \wedge u \in A\star \wedge v \in A\star$
    by *blast*
qed

lemma *finite-set-has-max2*:
  ⟦*finite* A; $A \neq \{\}$⟧ $\Longrightarrow \exists$ *max* $\in A$. $\forall$ $a \in A$. *length* $a \leq$ *length 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* {*xa*. *strict-prefix xa* (*x*::'*a list*)}
apply (*induct x rule*:*rev-induct*, *simp*)
apply (*subgoal-tac* {*xa*. *strict-prefix xa* (*xs* @ [*x*])} = {*xa*. *strict-prefix xa xs*} $\cup$ {*xs*})
by (*auto simp*:*strict-prefix-def*)

lemma *append-eq-cases*:
  assumes *a*: x @ $y = m$ @ n $m \neq []$
  shows *prefix* x $m \vee$ *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 $\exists (u, v) \in$ *Partitions* x. $\exists$ $(u', v') \in$ *Partitions* z. *strict-prefix* u $x \wedge u \in A\star \wedge v$ @ $u' \in A \wedge v' \in A\star$
proof –
  define S **where** $S = \{u \mid u\ v.\ (u, v) \in$ *Partitions* $x \wedge$ *strict-prefix* u $x \wedge u \in A\star \wedge v$ @ $z \in A\star\}$
  have *finite* {u. *strict-prefix* u x} **by** (*rule finite-strict-prefix-set*)
  then have *finite* S **unfolding** *S-def*
    by (*rule rev-finite-subset*) (*auto*)
  moreover
  have $S \neq \{\}$ **using** *a b* **unfolding** *S-def Partitions-def*
    by (*auto simp*: *strict-prefix-def*)
  ultimately have $\exists$ *u-max* $\in S$. $\forall$ $u \in S$. *length* $u \leq$ *length u-max*
    using *finite-set-has-max2* **by** *blast*

```
  then obtain u-max v
    where h0: (u-max, v) ∈ Partitions x
    and h1: strict-prefix u-max x
    and h2: u-max ∈ A⋆
    and h3: v @ z ∈ A⋆
    and h4: ∀ u v. (u, v) ∈ Partitions x ∧ strict-prefix u x ∧ u ∈ A⋆ ∧ v @ z ∈
A⋆ ⟶ length u ≤ length u-max
    unfolding S-def Partitions-def by blast
  have q: v ≠ [] using h0 h1 b unfolding Partitions-def by auto
  from h3 obtain a b
    where i1: (a, b) ∈ Partitions (v @ z)
    and   i2: a ∈ A
    and   i3: b ∈ A⋆
    and   i4: a ≠ []
    unfolding Partitions-def
    using q by (auto dest: star-decom)
  have prefix v a
  proof (rule ccontr)
    assume a: ¬(prefix v a)
    from i1 have i1': a @ b = v @ z unfolding Partitions-def by simp
    then have prefix a v ∨ strict-prefix v a using append-eq-cases q by blast
    then have q: strict-prefix a v using a unfolding strict-prefix-def prefix-def by
auto
    then obtain as where eq: a @ as = v unfolding strict-prefix-def prefix-def
by auto
    have (u-max @ a, as) ∈ Partitions x using eq h0 unfolding Partitions-def
by auto
    moreover
    have strict-prefix (u-max @ a) x using h0 eq q unfolding Partitions-def
prefix-def strict-prefix-def by auto
    moreover
    have u-max @ a ∈ A⋆ using i2 h2 by simp
    moreover
    have as @ z ∈ A⋆ using i1' i2 i3 eq by auto
    ultimately have length (u-max @ a) ≤ length u-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) ∈ Partitions z
    and   k4: zb = b
    unfolding Partitions-def prefix-def
    by (auto simp add: append-eq-append-conv2)
  show ∃ (u, v) ∈ Partitions x. ∃ (u', v') ∈ Partitions z. strict-prefix u x ∧ u ∈
A⋆ ∧ v @ u' ∈ A ∧ v' ∈ A⋆
    using h0 h1 h2 i2 i3 k1 k2 k4 unfolding Partitions-def by blast
qed

definition
```

```
  tag-Star :: 'a lang ⇒ 'a list ⇒ ('a lang) set
where
  tag-Star A ≡ λx. {≈A `` {v} | u v. strict-prefix u x ∧ u ∈ A⋆ ∧ (u, v) ∈ Partitions
x}

lemma tag-Star-non-empty-injI:
  assumes a: tag-Star A x = tag-Star A y
  and     c: x @ z ∈ A⋆
  and     d: x ≠ []
  shows y @ z ∈ A⋆
proof −
  obtain u v u' v'
    where a1: (u,  v) ∈ Partitions x (u', v')∈ Partitions z
    and   a2: strict-prefix u x
    and   a3: u ∈ A⋆
    and   a4: v @ u' ∈ A
    and   a5: v' ∈ A⋆
    using c d by (auto dest: star-spartitions-elim2)
  have (≈A) `` {v} ∈ 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 (≈A) `` {v} ∈ tag-Star A y using a by simp
  then obtain u1 v1
    where b1: v ≈A v1
    and   b3: u1 ∈ A⋆
    and   b4: (u1, v1) ∈ Partitions y
    unfolding tag-Star-def by auto
  have c: v1 @ u' ∈ A⋆ using b1 a4 unfolding str-eq-def by simp
  have u1 @ (v1 @ u') @ v' ∈ A⋆
    using b3 c a5 by (simp only: append-in-starI)
  then show y @ z ∈ A⋆ 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 ∈ A⋆
  and     d: x = []
  shows y @ z ∈ A⋆
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 ∈ A⋆ using c d by simp
qed

lemma quot-star-finiteI [intro]:
  assumes finite1: finite (UNIV // ≈A)
  shows finite (UNIV // ≈(A⋆))
proof (rule-tac tag = tag-Star A in tag-finite-imageD)
  have ⋀x y z. ⟦tag-Star A x = tag-Star A y; x @ z ∈ A⋆⟧ ⟹ y @ z ∈ A⋆
    by (case-tac x = []) (blast intro: tag-Star-empty-injI tag-Star-non-empty-injI)+
  then show =(tag-Star A)= ⊆ ≈(A⋆)
    by (rule refined-intro) (auto simp add: tag-eq-def)
next
  have ∗: finite (Pow (UNIV // ≈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 // ≈(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 (∃ r. A = lang r) ⟷ finite (UNIV // ≈A)
using Myhill-Nerode1 Myhill-Nerode2 by auto
```

4.1 Second direction proved using partial derivatives

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

```
lemma MN-Rel-Derivs:
  shows x ≈A y ⟷ Derivs x A = Derivs y A
unfolding Derivs-def str-eq-def
by auto

lemma Myhill-Nerode3:
  fixes r::'a rexp
  shows finite (UNIV // ≈(lang r))
```

```
proof -
  have finite (UNIV // =(λx. pderivs x r)=)
  proof -
    have range (λx. pderivs x r) ⊆ 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 (λx. pderivs x r))
      by (simp add: finite-subset)
    then show finite (UNIV // =(λx. pderivs x r)=)
      by (rule finite-eq-tag-rel)
  qed
  moreover
  have =(λx. pderivs x r)= ⊆ ≈(lang r)
    unfolding tag-eq-def
    by (auto simp add: MN-Rel-Derivs Derivs-pderivs)
  moreover
  have equiv UNIV =(λx. pderivs x r)=
  and  equiv UNIV (≈(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 // ≈(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 ⇒ bool
where
  regular A ≡ ∃ r. A = lang r
```

5.1 Closure under ∪, · and ⋆

```
lemma closure-union [intro]:
  assumes regular A regular B
  shows regular (A ∪ B)
proof -
  from assms obtain r1 r2::'a rexp where lang r1 = A lang r2 = B by auto
  then have A ∪ B = lang (Plus r1 r2) by simp
  then show regular (A ∪ B) by blast
```

qed

lemma *closure-seq* [*intro*]:
  assumes *regular A regular B*
  shows *regular* $(A \cdot B)$
proof −
  from *assms* **obtain** *r1 r2::'a rexp* **where** *lang r1* = *A lang r2* = *B* **by** *auto*
  then have $A \cdot B$ = *lang* (*Times r1 r2*) **by** *simp*
  then show *regular* $(A \cdot B)$ **by** *blast*
qed

lemma *closure-star* [*intro*]:
  assumes *regular A*
  shows *regular* $(A\star)$
proof −
  from *assms* **obtain** *r::'a rexp* **where** *lang r* = *A* **by** *auto*
  then have $A\star$ = *lang* (*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::finite) lang*
  assumes *regular A*
  shows *regular* (− *A*)
proof −
  from *assms* **have** *finite* (*UNIV* // ≈*A*) **by** (*simp add*: *Myhill-Nerode*)
  then have *finite* (*UNIV* // ≈(−*A*)) **by** (*simp add*: *str-eq-def*)
  then show *regular* (− *A*) **by** (*simp add*: *Myhill-Nerode*)
qed

5.3 Closure under − and ∩

lemma *closure-difference* [*intro*]:
  fixes *A::('a::finite) lang*
  assumes *regular A regular B*
  shows *regular* (*A* − *B*)
proof −
  have *A* − *B* = − (− *A* ∪ *B*) **by** *blast*
  moreover
  have *regular* (− (− *A* ∪ *B*))
    using *assms* **by** *blast*
  ultimately show *regular* (*A* − *B*) **by** *simp*
qed

lemma *closure-intersection* [*intro*]:
  fixes *A::('a::finite) lang*

```
  assumes regular A regular B
  shows regular (A ∩ B)
proof −
  have A ∩ B = − (− A ∪ − 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 rev ‘ (B · A) = (rev ‘ A) · (rev ‘ B)
unfolding conc-def image-def
by (auto) (metis rev-append)+

lemma rev-star1:
  assumes a: s ∈ (rev ‘ A)⋆
  shows s ∈ rev ‘ (A⋆)
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 ∈ rev ‘ A s2 ∈ rev ‘ (A⋆) by fact+
  then obtain x1 x2 where x1 ∈ A x2 ∈ A⋆ and eqs: s1 = rev x1 s2 = rev x2
by auto
  then have x1 ∈ A⋆ x2 ∈ A⋆ by (auto)
  then have x2 @ x1 ∈ A⋆ by (auto)
 then have rev (x2 @ x1) ∈ rev ‘ A⋆ using inj by (simp only: inj-image-mem-iff)
  then show s1 @ s2 ∈  rev ‘ A⋆ using eqs by simp
qed (auto)

lemma rev-star2:
  assumes a: s ∈ A⋆
  shows rev s ∈ (rev ‘ A)⋆
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* ∈ *A***by** *fact*
then have *rev s1* ∈ *rev* ‘ *A* **using** *inj* **by** (*simp only*: *inj-image-mem-iff*)
then have *rev s1* ∈ (*rev* ‘ *A*)⋆ **by** (*auto*)
moreover
have *rev s2* ∈ (*rev* ‘ *A*)⋆ **by** *fact*
ultimately show *rev* (*s1* @ *s2*) ∈ (*rev* ‘ *A*)⋆ **by** (*auto*)
qed (*auto*)

lemma *rev-star* [*simp*]:
shows *rev* ‘ (*A*⋆) = (*rev* ‘ *A*)⋆
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 ≡ ⋃*x* ∈ *A*. *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*) = (⋃ (*lang* ‘ *pderivs-lang B r*))
by (*simp add*: *Derivs-pderivs pderivs-lang-def*)
also have ... = *lang* (⨄(*pderivs-lang B r*)) **using** *fin* **by** *simp*
finally have *Deriv-lang B A* = *lang* (⨄(*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 $\{[]\} = \mathit{lang}\ (\mathit{One})$ **by** *simp*
  then show *regular* $\{[]\}$ **by** *blast*
next
  case (*Cons c s*)
  have *regular* $\{s\}$ **by** *fact*
  then obtain *r* **where** $\{s\} = \mathit{lang}\ r$ **by** *blast*
  then have $\{c\ \#\ s\} = \mathit{lang}\ (\mathit{Times}\ (\mathit{Atom}\ c)\ r)$
    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 $\{\} = \mathit{lang}\ (\mathit{Zero})$ **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 (‹- ^^^ -› [100, 100] 100)
where
  a ^^^ n ≡ replicate n a

lemma an-bn-not-regular:
  shows ¬ regular (⋃n. {CHR ''a'' ^^^ n @ CHR ''b'' ^^^ n})
proof
  define A where A = (⋃n. {CHR ''a'' ^^^ n @ CHR ''b'' ^^^ n})
  assume as: regular A
  define B where B = (⋃n. {CHR ''a'' ^^^ n})

  have sameness: ⋀i j. CHR ''a'' ^^^ i @ CHR ''b'' ^^^ j ∈ A ⟷ 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=λn. CHR ''a'' ^^^ n in exI) (auto)
  moreover
  have ∀ x ∈ B. ∀ y ∈ B. x ≠ y ⟶ ¬ (x ≈A y)
    apply(auto)
    unfolding B-def
    apply(auto)
    apply(simp add: str-eq-def)
    apply(drule-tac x=CHR ''b'' ^^^ 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 ≠ 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 ∧ x ≠ y}
proof –
  have wf (measure length) by simp
  moreover
  have {(x, y). subseq x y ∧ x ≠ y} ⊆ measure length
    unfolding measure-def by (auto simp add: subseq-strict-length)
  ultimately
  show wf {(x, y). subseq x y ∧ x ≠ y} by (rule wf-subset)
qed

lemma subseq-good:
  shows good subseq (f :: nat ⇒ ('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: ∀ (x::('a::finite) list) ∈ A. ∀ y ∈ A. x ≠ y ⟶ ¬(subseq x y) ∧
¬(subseq y x)
  shows finite A
proof (rule ccontr)
  assume infinite A
  then obtain f::nat ⇒ 'a::finite list where b: inj f and c: range f ⊆ A
    by (auto simp add: infinite-iff-countable-subset)
  from subseq-good[where f=f]
  obtain i j where d: i < j and e: subseq (f i) (f j) ∨ f i = f j
    unfolding good-def
    by auto
  have f i ≠ f j using b d by (auto simp add: inj-on-def)
  moreover
  have f i ∈ A using c by auto
  moreover
  have f j ∈ A using c by auto
  ultimately have ¬(subseq (f i) (f j)) using a by simp
  with e show False by auto
qed
```

6.1 Sub- and Supersequences

```
definition
 SUBSEQ A ≡ {x::('a::finite) list. ∃ y ∈ A. subseq x y}

definition
 SUPSEQ A ≡ {x::('a::finite) list. ∃ y ∈ A. 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 · {[c]} · UNIV
unfolding SUPSEQ-def conc-def
by (auto dest: list-emb-ConsD)

lemma SUPSEQ-union [simp]:
  shows SUPSEQ (A ∪ B) = SUPSEQ A ∪ SUPSEQ B
unfolding SUPSEQ-def by auto

lemma SUPSEQ-conc [simp]:
  shows SUPSEQ (A · B) = SUPSEQ A · 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*$\star$) = *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* ' *UNIV*)

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

lemma [*simp*]:
  shows ($\bigcup$ *a*. {[*a*]})$\star$ = *UNIV*
apply(*auto*)
apply(*induct-tac x*)
apply(*auto*)
apply(*subgoal-tac* [*a*] @ *list* $\in$ ($\bigcup$ *a*. {[*a*]})$\star$)
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::finite lang
  assumes regular A
  shows regular (SUPSEQ A)
proof -
  from assms obtain r::'a::finite rexp where lang r = A by auto
  then have lang (UP r) = SUPSEQ A by (simp add: lang-UP)
  then show regular (SUPSEQ A) by auto
qed

lemma SUPSEQ-subset:
  fixes A::'a::finite list set
  shows A ⊆ SUPSEQ A
unfolding SUPSEQ-def by auto

lemma SUBSEQ-complement:
  shows − (SUBSEQ A) = SUPSEQ (− (SUBSEQ A))
proof -
  have − (SUBSEQ A) ⊆ SUPSEQ (− (SUBSEQ A))
    by (rule SUPSEQ-subset)
  moreover
  have SUPSEQ (− (SUBSEQ A)) ⊆ − (SUBSEQ A)
  proof (rule ccontr)
    assume ¬ (SUPSEQ (− (SUBSEQ A)) ⊆ − (SUBSEQ A))
    then obtain x where
       a: x ∈ SUPSEQ (− (SUBSEQ A)) and
       b: x ∉ − (SUBSEQ A) by auto

    from a obtain y where c: y ∈ − (SUBSEQ A) and d: subseq y x
      by (auto simp add: SUPSEQ-def)

    from b have x ∈ SUBSEQ A by simp
    then obtain x' where f: x' ∈ A and e: subseq x x'
      by (auto simp add: SUBSEQ-def)

    from d e have subseq y x'
      by (rule subseq-order.order-trans)
    then have y ∈ SUBSEQ A using f
      by (auto simp add: SUBSEQ-def)
    with c show False by simp
  qed
  ultimately show − (SUBSEQ A) = SUPSEQ (− (SUBSEQ A)) by simp
qed

definition
  minimal :: 'a::finite list ⇒ 'a lang ⇒ bool
where
```

```
  minimal x A ≡ (∀ y ∈ A. subseq y x ⟶ subseq x y)

lemma main-lemma:
  shows ∃ M. finite M ∧ SUPSEQ A = SUPSEQ M
proof −
  define M where M = {x ∈ 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 ⊆ SUPSEQ M
  proof
    fix x
    assume x ∈ SUPSEQ A
   then obtain y where y ∈ A and subseq y x by (auto simp add: SUPSEQ-def)
    then have a: y ∈ {y′ ∈ A. subseq y′ x} by simp
     obtain z where b: z ∈ A subseq z x and c: ∀ y. subseq y z ∧ y ≠ z ⟶ y ∉
{y′ ∈ A. subseq y′ x}
      using wfE-min[OF subseq-wf a] by auto
    then have z ∈ M
      unfolding M-def minimal-def
      by (auto intro: subseq-order.order-trans)
    with b(2) show x ∈ SUPSEQ M
      by (auto simp add: SUPSEQ-def)
  qed
  moreover
  have SUPSEQ M ⊆ SUPSEQ A
    by (auto simp add: SUPSEQ-def M-def)
  ultimately
  show ∃ M. finite M ∧ 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 (λy. *f* −' {*y*}) (*f* ' *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* :: *finite rexp*
shows *finite* (*range* (λw. *Derivs w* (*lang r*)))
proof −
have *?thesis* $\longleftrightarrow$ *finite* (*UNIV* // $\approx$*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*: [] ∈ *Derivs w A* $\longleftrightarrow$ *w* ∈ *A*
by (*auto simp*: *Derivs-def*)

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

primrec *repeat* :: *nat* ⇒ *'a list* ⇒ *'a list* **where**
repeat 0 xs = []
| *repeat* (*Suc n*) *xs* = *xs* @ *repeat n xs*

lemma *repeat-Cons-left*: *repeat* (*Suc n*) *xs* = *xs* @ *repeat n xs* **by** *simp*

lemma *repeat-Cons-right*: *repeat* (*Suc n*) *xs* = *repeat n xs* @ *xs*
by (*induction n*) *simp-all*

lemma *repeat-Cons-append-commute* [*simp*]: *repeat n xs* @ *xs* = *xs* @ *repeat n xs*
by (*subst repeat-Cons-right* [*symmetric*]) *simp*

lemma *repeat-Cons-add* [*simp*]: *repeat* (*m* + *n*) *xs* = *repeat m xs* @ *repeat n xs*
by (*induction m*) *simp-all*

```
lemma repeat-Nil [simp]: repeat n [] = []
  by (induction n) simp-all

lemma repeat-conv-replicate: repeat n xs = concat (replicate n xs)
  by (induction n) simp-all
```

```
lemma nth-prefixes [simp]: n ≤ length xs ⟹ prefixes xs ! n = take n xs
  by (induction xs arbitrary: n) (auto simp: nth-Cons split: nat.splits)

lemma nth-suffixes [simp]: n ≤ length xs ⟹ suffixes xs ! n = drop (length xs −
n) xs
  by (subst suffixes-conv-prefixes) (simp-all add: rev-take)

lemma length-take-prefixes:
  assumes xs ∈ set (take n (prefixes ys))
  shows   length xs < n
proof (cases n ≤ Suc (length ys))
  case True
  with assms obtain i where i < n xs = take i ys
    by (subst (asm) nth-image [symmetric]) auto
  thus ?thesis by simp
next
  case False
  with assms have prefix xs ys by simp
  hence length xs ≤ length ys by (rule prefix-length-le)
  also from False have ... < 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 infinite B ⋀x y. x ∈ B ⟹ y ∈ B ⟹ x ≠ y ⟹ ∃ w. ¬(x @ w ∈ A
⟷ y @ w ∈ A)
  shows   ¬regular-lang (A :: 'a :: finite list set)
proof −
  have (λw. Derivs w A) ‘ B ⊆ range (λw. Derivs w A) by blast
  moreover from assms(2) have inj-on (λw. Derivs w A) B
    by (auto simp: inj-on-def Derivs-def)
  with assms(1) have infinite ((λw. Derivs w A) ‘ B)
    by (blast dest: finite-imageD)
  ultimately have infinite (range (λw. Derivs w A)) by (rule infinite-super)
  with regular-Derivs-finite show ?thesis by blast
qed
```

```
lemma not-regular-langI':
  assumes infinite B ⋀x y. x ∈ B ⟹ y ∈ B ⟹ x ≠ y ⟹ ∃ w. ¬(f x @ w ∈ A
⟷ f y @ w ∈ A)
  shows  ¬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 ‹infinite B› 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 δ ≡ λw. Derivs w A
  defines n ≡ card (range δ)
  assumes z ∈ A finite (range δ) length z ≥ n
  shows ∃ u v w. z = u @ v @ w ∧ length (u @ v) ≤ n ∧ v ≠ [] ∧ (∀ i. u @ repeat
i v @ w ∈ A)
proof −
  define P where P = set (take (Suc n) (prefixes z))
  from ‹length z ≥ n› have [simp]: card P = Suc n
    unfolding P-def by (subst distinct-card) (auto intro!: distinct-take)
  have length-le: length y ≤ n if y ∈ P for y
    using length-take-prefixes[OF that [unfolded P-def]] by simp

  have card (δ ` P) ≤ card (range δ) by (intro card-mono assms) auto
  also from assms have ... < card P by simp
  finally have ¬inj-on δ P by (rule pigeonhole)
  then obtain a b where ab: a ∈ P b ∈ P a ≠ b Derivs a A = 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 ≤ n length b ≤ n
    by (simp-all add: length-le)

  have ∗: ?thesis
    if uz': prefix u z' prefix z' z length z' ≤ n
           u ≠ z' Derivs z' A = Derivs u A for u z'
  proof −
    from ‹prefix u z'› and ‹u ≠ z'›
      obtain v where v [simp]: z' = u @ v v ≠ []
      by (auto simp: prefix-def)
    from ‹prefix z' z› 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 ‹z ∈ A› and uz' have ∀ i. u @ repeat i v @ w ∈ A
      by (simp add: Nil-in-Derivs-iff [of - A, symmetric])
    moreover have z = u @ v @ w by simp
    moreover from ‹length z' ≤ n› have length (u @ v) ≤ n by simp
    ultimately show ?thesis using ‹v ≠ []› by blast
  qed

  from prefix-ab have prefix a b ∨ 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 :: finite rexp
  obtains n where
    ⋀z. z ∈ lang r ⟹ length z ≥ n ⟹
           ∃ u v w. z = u @ v @ w ∧ length (u @ v) ≤ n ∧ v ≠ [] ∧ (∀ i. u @ repeat
i v @ w ∈ lang r)
proof −
  let ?n = card (range (λw. Derivs w (lang r)))
  have ∃ u v w. z = u @ v @ w ∧ length (u @ v) ≤ ?n ∧ v ≠ [] ∧ (∀ i. u @ repeat
i v @ w ∈ lang r)
    if z ∈ lang r and length z ≥ ?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 :: finite list set
  assumes ⋀n. length (z n) ≥ n and ⋀n. z n ∈ A
  assumes ⋀n u v w. z n = u @ v @ w ⟹ length (u @ v) ≤ n ⟹ v ≠ [] ⟹
           u @ repeat (i n u v w) v @ w ∉ A
  shows  ¬regular-lang A
proof
  assume regular-lang A
  then obtain r where r: lang r = A by blast
  from pumping-lemma[of r] obtain n
    where z n ∈ lang r ⟹ n ≤ length (z n) ⟹
      ∃ u v w. z n = u @ v @ w ∧ length (u @ v) ≤ n ∧ v ≠ [] ∧ (∀ i. u @ repeat i
v @ w ∈ lang r)
    by metis
  from this and assms[of n] obtain u v w
    where z n = u @ v @ w and length (u @ v) ≤ n and v ≠ [] and
        ⋀i. u @ repeat i v @ w ∈ 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*. *length* (*filter id w*) = *length* (*filter Not w*)} (**is** $\neg$*regular-lang ?A*)
proof (*rule not-regular-langI'*)
  show *infinite* (*UNIV* :: *nat set*) **by** *simp*
next
  fix *m n* :: *nat* **assume** *m* ≠ *n*
  hence *replicate m True* @ *replicate m False* ∈ *?A* **and**
        replicate n True @ *replicate m False* ∉ *?A* **by** *simp-all*
  thus ∃ *w*. ¬(*replicate m True* @ *w* ∈ *?A* ⟷ *replicate n True* @ *w* ∈ *?A*) **by** *blast*
qed

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

lemma *eq-replicate-iff*:
  xs = *replicate n x* ⟷ *set xs* ⊆ {*x*} ∧ *length xs* = *n*
  using *replicate-length-same*[*of xs x*] **by** (*subst eq-commute*) *auto*

lemma *replicate-eq-appendE*:
  assumes *xs* @ *ys* = *replicate n x*
  obtains *i j* **where** *n* = *i* + *j* *xs* = *replicate i x* *ys* = *replicate j x*
proof −
  have *n* = *length* (*replicate n x*) **by** *simp*
  also note *assms* [*symmetric*]
  finally have *n* = *length xs* + *length ys* **by** *simp*
  moreover have *xs* = *replicate* (*length xs*) *x* **and** *ys* = *replicate* (*length ys*) *x*
    using *assms* **by** (*auto simp*: *eq-replicate-iff*)
  ultimately show *?thesis* **using** *that*[*of length xs length ys*] **by** *auto*
qed

lemma $\neg$*regular-lang* (*range* (λ*i*. *replicate i True* @ *replicate i False*)) (**is** $\neg$*regular-lang ?A*)
proof (*rule pumping-lemma-not-regular-lang*)
  fix *n* :: *nat*
  show *length* (*replicate n True* @ *replicate n False*) ≥ *n* **by** *simp*
  show *replicate n True* @ *replicate n False* ∈ *?A* **by** *simp*
next
  fix *n* :: *nat* **and** *u v w* :: *bool list*
  assume *decomp*: *replicate n True* @ *replicate n False* = *u* @ *v* @ *w*
    and *length-le*: *length* (*u* @ *v*) ≤ *n* **and** *v-ne*: *v* ≠ []
  define *w1 w2* **where** *w1* = *take* (*n* − *length* (*u*@*v*)) *w* **and** *w2* = *drop* (*n* − *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 ‹*v* ≠ []› **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.