The Myhill-Nerode Theorem
Based on Regular Expressions
Chunhan Wu, Xingyuan Zhang and Christian Urban
August 16, 2018

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.\footnote{Most details of the theories are described in the paper \cite{2}.}

Contents

1 “Summation” for regular expressions 2

2 First direction of MN: finite partition \(\Rightarrow\) regular language 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 Interations .......................................................... 10
  2.7 The conclusion of the first direction .............................. 16

3 Second direction of MN: regular language \(\Rightarrow\) finite partition 18
  3.1 Tagging functions .................................................. 18
  3.2 Base cases: Zero, One and Atom ................................. 21
  3.3 Case for Plus ....................................................... 22
  3.4 Case for Times ..................................................... 22
  3.5 Case for Star ....................................................... 25
  3.6 The conclusion of the second direction ........................... 28
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

$folds :: (\forall a \ b \ set \ . \ b \Rightarrow a \ set \Rightarrow b)$

where

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

Plus-combination for a set of regular expressions

abbreviation

$Setalt :: \forall a rel \ set \ . \ a rel \ (\bigcup \setminus \ [1000] \ 999)$

where

$\bigcup A \equiv folds Plus Zero A$
For finite sets, $\text{Set}_\text{alt}$ is preserved under $\text{lang}$.

**lemma** folds-plus-simp [simp]:
fixes rs::('a rexp) set
assumes a: finite rs
shows $\text{lang} (\biguplus rs) = \bigcup (\text{lang} \cdot rs)$
unfolding folds-def
apply(rule set-eqI)
apply(rule someI2-ex)
apply(rule-tac finite-imp-fold-graph[OF a])
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
cconc (infixr $\cdot$ 100) and
estar ($\ast$ $[101]$ 102)

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

Myhill-Nerode relation

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

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). Derivs u A = Derivs v A\}
by (auto simp: str-eq-def Derivs-def)

definition
finals :: 'a lang $\Rightarrow$ 'a lang set
where
finals $A \equiv \{ \approx A \ ' s \ | s . s \in A \}$

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

lemma finals-in-partitions:
shows $\text{finals} A \subseteq (\text{UNIV } / / \approx A)$
unfolding finals-def quotient-def
by auto

2.1 Equational systems

The two kinds of terms in the rhs of equations.
datatype $\prime a \text{ trm} =$
  Lam $\prime a \text{ rexp}$
  | Trn $\prime a \text{ lang} \prime a \text{ rexp}$

fun
  lang-trm::$\prime a \text{ trm} \Rightarrow \prime a \text{ lang}$
where
  lang-trm (Lam $r$) = $\text{lang} r$
  | lang-trm (Trn $X \ r$) = $X \cdot \text{lang} r$

fun
  lang-rhs::($\prime a \text{ trm}) \text{ set} \Rightarrow \prime a \text{ lang}$
where
  lang-rhs rhs = $\bigcup (\text{lang-trm} \ ' \ rhs)$

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

lemma lang-rhs-union-distrib:
shows $\text{lang-rhs} A \cup \text{lang-rhs} B = \text{lang-rhs} (A \cup B)$
by simp

  Transitions between equivalence classes
definition
transition :: $\prime a \text{ lang} \Rightarrow \prime a \Rightarrow \prime a \text{ lang} \Rightarrow \text{bool}$ (- $\models \Rightarrow - [100,100,100] 100$)
where
  $Y \models c \Rightarrow X \equiv Y \cdot \{c\} \subseteq X$

Initial equational system
definition
Init-rhs CS X ≡ 
  if ([/] ∈ X) then 
  {Lam One} ∪ {Trn Y (Atom c) | Y c. Y ∈ CS ∧ Y |=c⇒ X} 
  else 
  {Trn Y (Atom c)| Y c. Y ∈ CS ∧ Y |=c⇒ X} 

definition 
Init CS ≡ {(X, Init-rhs CS X) | X. X ∈ CS} 

2.2 Arden Operation on equations 

fun 
  Append-rexp :: 'a rexp ⇒ 'a trm ⇒ '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 ≡ (Append-rexp rexp) rhs 

definition 
  Arden X rhs ≡ 
  Append-rexp-rhs (rhs − {Trn X r | r. Trn X r ∈ rhs}) (Star (∪ {r. Trn X r ∈ rhs})) 

2.3 Substitution Operation on equations 

definition 
  Subst rhs X xrhs ≡ 
  (rhs − {Trn X r | r. Trn X r ∈ rhs}) ∪ (Append-rexp-rhs xrhs (∪ {r. Trn X r ∈ rhs})) 

definition 
  Subst-all :: ('a lang × ('a trm) set) set ⇒ ('a lang ⇒ ('a trm) set ⇒ ('a lang × ('a trm) set) set) 
where 
  Subst-all ES X xrhs ≡ {(Y, Subst yrhs X xrhs) | Y yrhs. (Y, yrhs) ∈ ES} 

definition 
  Remove ES X xrhs ≡ 
  Subst-all (ES − {(X, xrhs)}) X (Arden X xrhs) 

2.4 While-combinator and invariants 

definition 
  Iter X ES ≡ (let (Y, yrhs) = SOME (Y, yrhs). (Y, yrhs) ∈ ES ∧ X ≠ Y 
  in Remove ES Y yrhs) 

lemma IterI2:
assumes \((Y, \text{yrhs}) \in \text{ES}\)
and \(X \neq Y\)
and \(\bigwedge Y \text{yrhs}. \left[ (Y, \text{yrhs}) \in \text{ES}; X \neq Y \right] \implies Q\) (Remove \(\text{ES} Y \text{yrhs}\))

shows \(Q\) (Iter \(X \text{ES}\))

unfolding Iter-def using assms
by (rule-tac \(a=(Y, \text{yrhs})\) in someI2) (auto)

abbreviation
\(\text{Cond} \text{ES} \equiv \text{card} \text{ES} \neq 1\)

definition
\(\text{Solve} X \text{ES} \equiv \text{while} \text{Cond} (\text{Iter} X) \text{ES}\)

definition
distinctness \(\text{ES} \equiv \forall X \text{rhs} \text{rhs}'. (X, \text{rhs}) \in \text{ES} \wedge (X, \text{rhs}') \in \text{ES} \implies \text{rhs} = \text{rhs}'\)

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

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

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

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

lemma finite-rhs-def2:
\(\text{finite-rhs} \text{ES} = (\forall X \text{rhs}. (X, \text{rhs}) \in \text{ES} \implies \text{finite} \text{rhs})\)

unfolding finite-rhs-def by auto

definition
\(\text{rhss} \text{rhs} \equiv \{ X \mid X r. \text{Trn} X r \in \text{rhs} \}\)

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

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

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 \((auto\ simp\ add: \text{lhss-def})\)

**definition**

\[
\text{invariant } ES \equiv \text{finite } ES \\
\land \text{finite-rhs } ES \\
\land \text{soundness } ES \\
\land \text{distinctness } ES \\
\land \text{ardenable-all } ES \\
\land \text{validity } ES
\]

**lemma** \text{invariantI}:

**assumes** soundness ES finite ES distinctness ES ardenable-all ES finite-rhs ES validity ES

**shows** invariant ES

**using** assms by \((simp\ add: \text{invariant-def})\)

**declare** [[\text{simproc add: finite-Collect}]]

**lemma** \text{finite-Trn}:

**assumes** fin: finite rhs

**shows** finite \(\{r. \text{Trn} Y r \in rhs\}\)

**using** assms by \((auto\ intro!: \text{finite-vimageI} simp\ add: \text{inj-on-def})\)

**lemma** \text{finite-Lam}:

**assumes** fin: finite rhs

**shows** finite \(\{r. \text{Lam} r \in rhs\}\)

**using** assms by \((auto\ intro!: \text{finite-vimageI} simp\ add: \text{inj-on-def})\)

**lemma** \text{trm-soundness}:

**assumes** finite:finite rhs

**shows** lang-rhs \((\{\text{Trn} X r | r. \text{Trn} X r \in rhs\}) = X \cdot (\text{lang} (\bigsqcup \{r. \text{Trn} X r \in rhs\}))\)

**proof** –

**have** finite \(\{r. \text{Trn} X r \in rhs\}\)

**by** \((\text{rule finite-Trn[OF finite]}))

**then show** lang-rhs \((\{\text{Trn} X r | r. \text{Trn} X r \in rhs\}) = X \cdot (\text{lang} (\bigsqcup \{r. \text{Trn} X r \in rhs\}))\)

**by** \((\text{simp\ only: lang-rhs-set lang-trm.simps})\ (auto\ simp\ add: \text{conc-def})\)

**qed**

**lemma** \text{lang-of-append-rexp}:

\(\text{lang-trm} (\text{Append-rexp} r \text{ trm}) = \text{lang-trm} \text{ trm} \cdot \text{lang} r\)

**by** \((\text{induct\ rule: \text{Append-rexp.induct}})\)

\((\text{auto\ simp\ add: \text{conc-assoc}})\)

**lemma** \text{lang-of-append-rexp-rhs}:
langs (Append-rexp-rhs rhs r) = langs rhs · langs r

unfolding Append-rexp-rhs-def
by (auto simp add: conc-def langs-of-append-rexp)

2.5 Initial 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 = langs rhs
proof
  show X ⊆ langs rhs
proof
    fix x
    assume in-X: x ∈ X
    { assume empty: x = []
      then have x ∈ langs rhs using X-in-eqs in-X
        unfolding Init-def Init-rhs-def
        by auto
    }
moreover
    { assume not-empty: x ≠ []

8
then obtain $s \ c$ where $\text{decom: } x = s \ @ \ [c]$

using rev-cases by blast

have $X \in \text{UNIV} / \approx A$ using $\text{X-in-eqs unfolding Init-def by auto}$

then obtain $Y$ where $Y \in \text{UNIV} / \approx A \ Y \cdot \{[c]\} \subseteq X \ s \in Y$

using $\text{decom in-X every-eqclass-has-transition by metis}$

then have $x \in \text{lang-rhs} \ \{\text{Trn} \ Y \ (\text{Atom} \ c)\} \ Y \ c. \ Y \in \text{UNIV} / \approx A \land Y$

$\models c \Rightarrow X$

unfolding $\text{transition-def}$

using $\text{decom by (force simp add: conc-def)}$

then have $x \in \text{lang-rhs rhs using X-in-eqs in-X}$

unfolding $\text{Init-def Init-rhs-def by simp}$

ultimately show $x \in \text{lang-rhs rhs by blast}$

qed

next

show $\text{lang-rhs rhs} \subseteq X$ using $\text{X-in-eqs}$

unfolding $\text{Init-def Init-rhs-def transition-def}$

by auto

qed

lemma $\text{finite-Init-rhs}$:

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

assumes $\text{finite-CS}$: finite $\ (\text{UNIV} / \approx A)$

shows $\text{finite} (\text{Init-rhs} \ CS \ X)$

using $\text{assms unfolding Init-rhs-def transition-def by simp}$

lemma $\text{Init-ES-satisfies-invariant}$:

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

assumes $\text{finite-CS}$: finite $\ (\text{UNIV} / \approx A)$

shows $\text{invariant} (\text{Init} \ (\text{UNIV} / \approx A))$

proof (rule invariantI)

show $\text{soundness} (\text{Init} \ (\text{UNIV} / \approx A))$

unfolding $\text{soundness-def}$

using $\text{l-eq-r-in-eqs by auto}$

show $\text{finite} (\text{Init} \ (\text{UNIV} / \approx A))$ using $\text{finite-CS}$

unfolding $\text{Init-def by simp}$

show $\text{distinctness} (\text{Init} \ (\text{UNIV} / \approx A))$

unfolding $\text{distinctness-def Init-def by simp}$

show $\text{ardenable-all} (\text{Init} \ (\text{UNIV} / \approx A))$

unfolding $\text{ardenable-all-def Init-def Init-rhs-def ardenable-def}$

by auto

show $\text{finite-rhs} (\text{Init} \ (\text{UNIV} / \approx A))$

using $\text{finite-Init-rhs[of finite-CS]}$

unfolding $\text{finite-rhs-def Init-def by auto}$

show $\text{validity} (\text{Init} \ (\text{UNIV} / \approx A))$

unfolding $\text{validity-def Init-def Init-rhs-def rhss-def lhss-def}$

by auto
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′] \implies ardenable (rhs \cup rhs′)
by (auto simp: ardenable-def)

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

lemma Subst-preserves-ardenable:
[ardenable rhs; ardenable xrhs] \implies 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 (⨄{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 (⨄{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] \implies 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-preserve-finite-rhs:
  finite-rhs ES; finite yrhs \implies 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-cl:
  rhss (Append-rxp-rhs rhs x) = rhss rhs
apply (auto simp: Append-rxp-rhs-def)
also (case-tac xa, auto simp: image-def)
by (rule-tac x = Times ra xa 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-cl)
by (auto simp: rhss-def)

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

lemma Subst-updates-cl:
  X /∈ rhss \implies rhss (Subst rhs X xrhs) = rhss rhs \union rhss xrhs - {X}
apply (simp only: Subst-def append-rhs-preserves-cl rhss-union-distrib)
by (auto simp: rhss-def)

lemma Subst-all-preserve-validity:
  assumes sc: validity (ES \union \{(Y, 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 xrhs
    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 \union 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-cl)
qed
  moreover have rhss xrhs \subseteq lhss ES \union \{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)
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: invariant-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 \( \text{card } ES > 1 \) using \( \text{card } e \)-in finite
by (cases card ES) (auto)
then have \( \text{card } (ES - \{(X, \text{rhs})\}) > 0 \)
using finite e-in by auto
then have \( (ES - \{(X, \text{rhs})\}) \neq \{\} \) using finite by (rule-tac notI, simp)
then show \( \exists (Y, \text{yrhs}) \in ES. (X, \text{rhs}) \neq (Y, \text{yrhs}) \)
by auto
qed

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

by \((\text{rule-tac } \text{Subst-all-satisfies-invariant}) \ (\text{simp})\)

qed

qed

lemma \textit{iteration-step-ex}:
assumes \(\text{Inv-ES}: \text{invariant } \text{ES}\)
and \(\text{X-in-ES}: (X, \text{xrhs}) \in \text{ES}\)
and \(\text{Cnd}: \text{Cond } \text{ES}\)
shows \(\exists \text{xrhs}' . (X, \text{xrhs}') \in (\text{Iter } X \text{ES})\)

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

qed

2.7 The conclusion of the first direction

lemma \textit{Solve}:
fixes \(A::(\text{a::finite lang})\)
assumes \(\text{fin}: \text{finite } (\text{UNIV } // \approx A)\)
and \(\text{X-in}: X \in (\text{UNIV } // \approx A)\)
shows \(\exists \text{rhs}. \text{Solve } X (\text{Init } (\text{UNIV } // \approx A)) = \{(X, \text{rhs})\} \land \text{invariant } \{(X, \text{rhs})\}\)

proof –
define \(\text{Inv}\) where \(\text{Inv } \text{ES} \leftarrow \text{invariant } \text{ES} \land (\exists \text{rhs}. (X, \text{rhs}) \in \text{ES})\)\ for \(\text{ES}\)
have \(\text{Inv } (\text{Init } (\text{UNIV } // \approx A))\) unfolding \(\text{Inv-def}\)
  using \(\text{fin } \text{X-in}\) by \((\text{simp add: Init-ES-satisfies-invariant, simp add: Init-def})\)
moreover
\{ fix \(\text{ES}\)
  assume \(\text{inv}: \text{Inv } \text{ES}\) and \(\text{crd}: \text{Cond } \text{ES}\)
  then have \(\text{Inv } (\text{Iter } X \text{ES})\)
    unfolding \(\text{Inv-def}\)
    by \((\text{auto simp add: iteration-step-invariant iteration-step-ex})\) \}
moreover
\{ fix \(\text{ES}\)
  assume \(\text{inv}: \text{Inv } \text{ES}\) and \(\text{not-crd}: \neg \text{Cond } \text{ES}\)
  from \(\text{inv}\) obtain \(\text{rhs}\) where \((X, \text{rhs}) \in \text{ES}\) unfolding \(\text{Inv-def}\) by \(\text{auto}\)
moreover
from \(\text{not-crd}\) have \(\text{card } \text{ES} = 1\) by \(\text{simp}\)
ultimately
have \( ES = \{ (X, \text{rhs}) \} \) by (auto simp add: card-Suc-eq)
then have \( \exists \text{rhs}'. \; ES = \{ (X, \text{rhs}') \} \wedge \text{invariant} \{(X, \text{rhs}')\} \) using inv
unfolding Inv-def by auto
moreover
have \( \exists \text{rhs}'. \; ES = \{ (X, \text{rhs}')\} \wedge \text{invariant} \{(X, \text{rhs}')\} \) using inv
unfolding Inv-def by auto
qed

lemma every-eqcl-has-reg:
  fixes \( A :: ('a::finite) \text{lang} \)
  assumes finite-CS: finite (UNIV // \approx A)
  and X-in-CS: \( X \in (\text{UNIV} // \approx A) \)
  shows \( \exists r. \; X = \text{lang \( r \)} \)
proof
  from finite-CS X-in-CS
  obtain \( \text{rhs} \) where Inv-ES: \( \text{invariant} \{(X, \text{rhs})\} \)
  using Solve by metis
  define \( A \) where \( A = \text{Arden X xrhs} \)
  have rhss xrhs \( \subseteq \{ X \} \) 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: \{\( \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. \; \text{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 $X = \text{lang-rhs} \{ \text{Lam } r \mid r. \text{Lam } r \in A \}$ using eq by simp
then have $X = \text{lang} (\bigcup \{ r. \text{Lam } r \in A \})$ 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 $\exists ys. (\bigcup S) = \bigcup (f ' ys) \land \text{finite } ys$
using bchoice[OF a] b
apply(erule-tac exE)
apply(rule-tac x = fa ' S in exI)
apply(auto)
done

theorem Myhill-Nerode1:
  fixes $A::('a::finite) \text{lang}$
  assumes finite-CS: finite (UNIV // ≈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 $\forall X \in (\text{UNIV // ≈A}). \exists r. X = \text{lang } r$
    using finite-CS every-eqcl-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 $rs::('a \text{ rexp}) \text{ set where } \bigcup \text{finals } A = \bigcup (\text{lang } rs) \text{ finite } rs$
    using fin by (auto dest: bchoice-finite-set)
  then have $A = \text{lang } (\bigcup rs)$
    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 ⇒ finite partition

3.1 Tagging functions

definition tag-eq :: ('a list ⇒ 'b) ⇒ ('a list × 'a list) set (=−=)
where
  (=−=) ≡ {(x, y). tag x = tag y}
abbreviation
tag-eq-applied :: 'a list ⇒ ('a list ⇒ 'b) ⇒ 'a list ⇒ bool (· =· ·)
where
  x· =· y ≡ (x, y) ∈ · =·

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

lemma refined-intro:
  assumes ⋀ x y z. [x· =· y; x@ z ∈ A] ←→ y@ z ∈ A
  shows =· =· ⊆ ≈ A
using assms unfolding str-eq-def tag-eq-def
apply (clarify, simp (no-asn-use))
by metis

lemma finite-eq-tag-rel:
  assumes rng-fnt: finite (range tag)
  shows finite (UNIV // =·)
proof −
  let ?f = λX. tag ' X and ?A = (UNIV // =·)
  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
    }
ultimately show finite \((\text{UNIV} \parallel =\text{tag}=)\) by (rule finite-imageD)

qed

lemma refined-partition-finite:
assumes fnt: finite (UNIV \parallel R1)
and refined: \(R1 \subseteq R2\)
and eq1: equiv UNIV R1 and eq2: equiv UNIV R2
shows finite (UNIV \parallel R2)

proof –
let \(\lambda X. \{ R1 \{ x \} \mid x \in X \}\)
and \(?A = \text{UNIV} \parallel R2\) and \(?B = \text{UNIV} \parallel R1\)
have \(\lambda ?A \subseteq \text{Pow} ?B\)
unfolding image-def Pow-def quotient-def by auto

moreover
have finite (\text{Pow} ?B) using fnt by simp

ultimately
have finite (\(\lambda ?A \subseteq \text{Pow} ?B\)) by (rule finite-subset)

moreover
have inj-on \(\lambda ?A \subseteq \text{Pow} ?B\) by (rule finite-subset)

qed

lemma tag-finite-imageD:
assumes rng-fnt: finite (range tag)
and refined: \(=\text{tag}= \subseteq \approx A\)
shows finite (UNIV \parallel \approx A)

proof (rule-tac refined-partition-finite [of \(=\text{tag}=\)])
show finite (UNIV \parallel \approx A) by (rule finite-eq-tag-rel[OF rng-fnt])

next
show \(=\text{tag}= \subseteq \approx A\) using refined .

qed
next  
  show  equiv UNIV =tag=  
  and  equiv UNIV (≈A)  
  unfolding equiv-def str-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 –
\{ \text{assume } y = [] \text{ hence } x = ([])} \text{ using } h \\
\quad \text{by (auto simp: str-eq-def)} \}

moreover
\{ \text{assume } y = [c] \text{ hence } x = ([c]) \text{ using } h \\
\quad \text{by (auto dest: spec[where } x = [] \text{ simp: str-eq-def})} \}

moreover
\{ \text{assume } y \neq [] \text{ and } y \neq [c] \text{ hence } \forall z. \, (y @ z) \neq [c] \text{ by (case-tac y, auto)} \\
\quad \text{moreover have } \bigwedge p. \, (p \neq [] \land p \neq [c]) = (\forall q. \, p @ q \neq [c]) \\
\quad \quad \text{by (case-tac p, auto)} \\
\quad \text{ultimately have } x = UNIV - {[]} - {[c]} \text{ using } h \\
\quad \quad \text{by (auto simp add: str-eq-def)} \}

ultimately show \ ?thesis by blast 
qed

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

3.3 Case for Plus

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

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

3.4 Case for Times

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

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

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

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

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

\text{lemma} \text{tag-Times-injI:}
\text{assumes} a: \text{tag-Times} A B x = \text{tag-Times} A B y
\text{and} c: x @ z \in A \cdot B
\text{shows} y @ z \in A \cdot B
\text{proof} –
\text{from c obtain u v where}
\text{h1:} (u, v) \in \text{Partitions } (x @ z) \text{ and}
\text{h2:} u \in A \text{ and}
\text{h3:} v \in B \text{ by (auto dest: conc-partitions-elim)}
\text{from h1 have} x @ z = u @ v \text{ unfolding \text{Partitions-def} by simp}
\text{then obtain us}
\text{where} (x = u @ us \land us @ z = v) \lor (x @ us = u \land z = us @ v)
\text{by (auto simp add: append-eq-append-conv2)}
\text{moreover}
\{ \text{ assume eq:} x = u @ us \land us @ z = v
\text{ have} (\approx B \{us\}) \in \text{snd} (\text{tag-Times} A B x)
\text{ unfolding \text{Partitions-def} \text{tag-Times-def} using h2 eq}
\text{ by (auto simp add: str-eq-def)}
\text{then have} (\approx B \{us\}) \in \text{snd} (\text{tag-Times} A B y)
\text{ using a by simp}
\text{then obtain} u' \text{us' where}
q1: u' \in A \text{ and}
q2: \approx B \{us\} = \approx B \{us'\} \text{ and}
\[ q_3: (u', us') \in \text{Partitions} y \]
unfolding \text{tag-Times-def} by auto
from \( q_2 h_3 eq \)
have \( us' \odot z \in B \)
unfolding \text{Image-def str-eq-def} by auto
then have \( y \odot z \in A \cdot B \) using \( q_1 q_3 \)
unfolding \text{Partitions-def} by auto
\}
moreover
\{ 
assume \( eq: \ x \odot us = u \odot z = us \odot v \)
have \( (\approx A \ '' \{x\}) = \text{fst} (\text{tag-Times} A B x) \)
by (simp add: \text{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: \text{tag-Times-def})
moreover
have \( x \odot us \in A \) using \( h_2 eq \) by simp
ultimately
have \( y \odot us \in A \) using \( h_2 eq \) by simp
unfolding \text{Image-def str-eq-def} by blast
then have \( (y \odot us) \odot v \in A \cdot B \)
using \( h_3 \) unfolding \text{conc-def} by blast
then have \( y \odot z \in A \cdot B \) using \( eq \) by simp
\}
ultimately show \( y \odot z \in A \cdot B \) by blast
qed

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

**Lemma star-partitions-elim:**

assumes \( x @ z \in A \ast, x \neq [] \)

shows \( \exists (u, v) \in \text{Partitions } (x @ z), \text{ strict-prefix } u x \land u \in A \ast \land v \in A \ast \)

**Proof** –

have \( ([], x @ z) \in \text{Partitions } (x @ z), \text{ strict-prefix } [] x [] \in A \ast x @ z \in A \ast \)

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

then show \( \exists (u, v) \in \text{Partitions } (x @ z), \text{ strict-prefix } u x \land u \in A \ast \land v \in A \ast \)

by blast

**QED**

**Lemma finite-set-has-max2:**

\([\text{finite } A; A \neq \{\} \implies \exists \text{ max } \in A. \forall a \in A. \text{ length } a \leq \text{ length } \text{ max} \)

**Applying** (induct rule: finite.induct)

**Applying** (simp)

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

**Lemma finite-strict-prefix-set:**

shows finite \( \{ u @ v. \text{ strict-prefix } u x (x :: 'a list) \} \)

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

**Applying** (subgoal-tac \( \{ u @ v. \text{ strict-prefix } u x (xs @ [x]) \} = \{ u. \text{ strict-prefix } u x xs \} \cup \{ xs \} \))

by (auto simp: strict-prefix-def)

**Lemma append-eq-cases:**

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

shows prefix \( x m \lor \text{ strict-prefix } m x \)

**Unfolding** prefix-def strict-prefix-def using \( a \)

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

**Lemma star-partitions-elim2:**

assumes \( a: x @ z \in A \ast \)

and \( b: x \neq [] \)

shows \( \exists (u, v) \in \text{Partitions } x. \exists (u', v') \in \text{Partitions } z. \text{ strict-prefix } u x \land u \in A \ast \land v @ u' \in A \land v' \in A \ast \)

**Proof** –

define \( S \) where \( S = \{ u | u v. (u, v) \in \text{Partitions } x \land \text{ strict-prefix } u x \land u \in A \ast \land v @ z \in A \ast \} \)

have finite \( \{ u. \text{ strict-prefix } u x \} \) 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 Partsitions-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 \( h2: u \text{-max} \in A^* \)
and \( h3: v \otimes z \in A^* \)
and \( h4: \forall u v. (u, v) \in \text{Partitions } x \land \text{strict-prefix } u x \land u \in A^* \land v \otimes z \in A^* \rightarrow \text{length } u \leq \text{length } u \text{-max} \)

**unfolding**

\( S \text{-def Partitions-def by blast} \)

have \( q: v \neq [] \) using \( h0 \) \( h1 \) \( b \)

**unfolding**

\( \text{Partitions-def by auto} \)

from \( h3 \) obtain \( a \) \( b \)

where \( i1: (a, b) \in \text{Partitions } (v \otimes z) \)
and \( i2: a \in A \)
and \( i3: b \in A^* \)
and \( i4: a \neq [] \)

**unfolding**

\( \text{Partitions-def by auto} \)

using \( q \) by (auto dest: star-decom)

have \( \text{prefix } v a \)

**proof**

(rule ccontr)

assume \( a: \neg(\text{prefix } v a) \)

from \( i1 \) have \( i1': a \otimes b = v \otimes z \) **unfolding**

\( \text{Partitions-def by simp} \)

then have \( \text{prefix } a v \lor \text{strict-prefix } v a \) 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 \( eq: a \otimes as = v \) **unfolding**

\( \text{strict-prefix-def prefix-def by auto} \)

by auto

have \( (u \text{-max } @ a, as) \in \text{Partitions } x \) using \( eq h0 \) **unfolding**

\( \text{Partitions-def by auto} \)

moreover

have \( \text{strict-prefix } (u \text{-max } @ a) x \) using \( h0 eq q \) **unfolding**

\( \text{Partitions-def by auto} \)

\( \text{strict-prefix-def prefix-def by auto} \)

moreover

have \( u \text{-max } @ a \in A^* \) using \( i2 h2 \) by simp

moreover

have \( as \otimes z \in A^* \) 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 \( k1: v \otimes za = a \)
and \( k2: (za, zb) \in \text{Partitions } z \)
and \( k4: zb = b \)

**unfolding**

\( \text{Partitions-def prefix-def by (auto simp add: append-eq-append-conv2)} \)

show \( \exists (u, v) \in \text{Partitions } x. \exists (u', v') \in \text{Partitions } z. \text{strict-prefix } u x \land u \in A^* \land v \otimes u' \in A \land v' \in A^* \)

using \( h0 h1 h2 i2 i3 k1 k2 k4 \) **unfolding**

\( \text{Partitions-def by blast} \)

qed

definition

\( \text{tag-Star } :: 'a \text{ lang } \Rightarrow 'a \text{ list } \Rightarrow ('a \text{ lang}) \text{ set} \)

where

\( \text{tag-Star } A \equiv \lambda x. \{ u. \text{strict-prefix } u x \land u \in A^* \land (u, v) \in \text{Partitions} \}

\text{ } \text{set} \)
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*

27
by (case-tac x = []) (blast intro: tag-Star-empty-injI tag-Star-non-empty-injI)+
then show \((\text{tag-Star}\ A)= \subseteq \approx (A^*)\)
  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 (\(\exists r.\ A = \text{lang } r\) \iff 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 \(x \approx A\ y \iff \text{Derivs } x A = \text{Derivs } y A\)
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 // \approx (\lambda x. \pderivs x r))
proof –
have range \(\lambda x. \text{pderivs } x \ r\) \(\subseteq\) 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 (\(\lambda x. \text{pderivs } x \ r\)))
by (simp add: finite-subset)
then show finite (UNIV // \(\lambda x. \text{pderivs } x \ r\))
by (rule finite-eq-tag-rel)
qed
moreover
have (\(\lambda x. \text{pderivs } x \ r\)) \(\subseteq\) \(\approx\) (lang \ r)
unfolding tag-eq-def by (auto simp add: MN-Rel-Derivs Derivs-pderivs)
moreover
have equiv UNIV (\(\lambda x. \text{pderivs } x \ r\))
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 \(\ast\)

lemma closure-union [intro]:
assumes regular A regular B
shows regular (A \(\cup\) B)
proof –
from assms obtain r1 r2::'a rexp 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 $(A \cdot B)$
proof –
  from assum obtain $r_1 r_2::\text{a rexp where}$ \hspace{1em} \begin{align*}
    \text{lang } r_1 &= A \\
    \text{lang } r_2 &= B
  \end{align*}
  by auto
then have $A \cdot B = \text{lang } (\text{Times } r_1 r_2)$ by simp
then show regular $(A \cdot B)$ by blast
qed

lemma closure-star [intro]:
assumes regular $A$
shows regular $(A \star)$
proof –
  from assum obtain $r::\text{a rexp where}$ \hspace{1em} \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 $A::(\text{finite lang})$
assumes regular $A$
shows regular $(- A)$
proof –
  from assum have finite $(\text{UNIV } \# A)$ by simp add: Myhill-Nerode
then have finite $(\text{UNIV } \# (- A))$ 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 $A::(\text{finite lang})$
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 assum by blast
ultimately show regular $(A - B)$ by simp
qed

lemma closure-intersection [intro]:
fixes $A::(\text{finite lang})$
assumes regular $A$ regular $B$
shows regular $(A \cap B)$
proof –

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

5.4 Closure under string reversal

fun
  $Rev :: 'a rexp \Rightarrow 'a rexp$
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 $rev \cdot (B \cdot A) = (rev \cdot A) \cdot (rev \cdot B)$
unfolding conc-def image-def
by (auto) (metis rev-append)+

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

lemma rev-star2:
  assumes a: $s \in A^*$
  shows $rev \ s \in (rev \cdot A)^*$
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 $s1 \in A\^$ fact
  then have $rev \ s1 \in rev \cdot A$ using inj by (simp only: inj-image-mem-iff)
then have \( \text{rev } s1 \in (\text{rev } A)^* \) by (auto)
mmoreover
have \( \text{rev } s2 \in (\text{rev } A)^* \) by fact
ultimately show \( \text{rev } (s1 \& s2) \in (\text{rev } A)^* \) by (auto)
qed (auto)

**Lemma**: \text{rev-star} [simp]:

\( \text{shows } \text{rev } (A^*) = (\text{rev } A)^* \)

**Using**: \text{rev-star1} \text{ rev-star2} by auto

**Lemma**: \text{rev-lang}:

\( \text{shows } \text{rev } (\text{lang } r) = \text{lang } (\text{Rev } r) \)
by (induct \( r \)) (simp-all add: image-Un)

**Lemma**: \text{closure-reversal} [intro]:

assumes regular \( A \)
shows regular \( (\text{rev } A) \)
proof –
from assms obtain \( r::'a \text{ rexp} \) where \( A = \text{lang } r \) by auto
then have \( \text{lang } (\text{Rev } r) = \text{rev } A \) by (simp add: rev-lang)
then show regular \( \text{rev } A \) by blast
qed

### 5.5 Closure under left-quotients

**Abbreviation**

\( \text{Deriv-lang } A B \equiv \bigcup x \in A. \text{Deriv } x B \)

**Lemma**: \text{closure-left-quotient}:

assumes regular \( A \)
shows regular \( (\text{Deriv-lang } B A) \)
proof –
from assms obtain \( r::'a \text{ rexp} \) where \( A = \text{lang } r \) by auto
have \( \text{fin} \): finite \( (\text{pderivs-lang } B r) \) by (rule finite-pderivs-lang)

have \( \text{Deriv-lang } B (\text{lang } r) = (\bigcup (\text{lang } \text{pderivs-lang } B r)) \)
by (simp add: Derivs-pderivs pderivs-lang-def)
also have \( . . . = \text{lang } (\bigcup (\text{pderivs-lang } B r)) \) using fin by simp
finally have \( \text{Deriv-lang } B A = \text{lang } (\bigcup (\text{pderivs-lang } B r)) \) using eq
by simp
then show regular \( (\text{Deriv-lang } B A) \) by auto
qed

### 5.6 Finite and co-finite sets are regular

**Lemma**: \text{singleton-regular}:

shows regular \( \{s\} \)
proof (induct \( s \))
case Nil
have \( \{\} = \text{lang } (\text{One}) \) by simp
then show regular \{[]\} 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\} ∪ 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 ∈ B. \exists y ∈ B. x \neq y ∧ \approx A y
proof
  define eqfun where eqfun = (λA x::('a::finite list). (≈ A) ‘\{x\})
  have finite (\text{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} 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 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
\( a `~` ~ n \equiv \text{replicate } n a \)

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: \( \bigwedge i j. \text{CHR } "a" `~` ~ i @ \text{CHR } "b" `~` ~ j \in A \leftrightarrow i = j \)
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" \ldots 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 (drule-tac x=CHR "b" \ldots 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.
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 $a$: $\forall (x::('a::finite) \text{ list}) \in A. \forall y \in A. x \neq y \rightarrow \neg (\text{subseq } x \ y) \land 
\neg (\text{subseq } y \ x)$
shows finite $A$

proof (rule ccontr)
assume infinite $A$
then obtain $f::\text{nat} \Rightarrow 'a::finite \text{ list}$
where $b$: inj $f$ and $c$: 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 $e$: subseq $(f \ i) (f \ j) \lor f \ i = f \ j$
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
$\text{SUBSEQ } A \equiv \{x::('a::finite) \text{ list}. \exists y \in A. \text{subseq } x \ y\}$

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

lemma SUPSEQ-simps [simp]:
shows $\text{SUPSEQ } \{\}$ = $\{\}$
and $\text{SUPSEQ } \{[]\} = \text{UNIV}$
unfolding SUPSEQ-def by auto

lemma SUPSEQ-atom [simp]:
shows $\text{SUPSEQ } \{[c]\} = \text{UNIV} \cdot \{[c]\} \cdot \text{UNIV}$
unfolding SUPSEQ-def conc-def
by (auto dest: list-emb-ConsD)

lemma SUPSEQ-union [simp]:
shows $\text{SUPSEQ } (A \cup B) = \text{SUPSEQ } A \cup \text{SUPSEQ } B$
unfolding SUPSEQ-def by auto

lemma SUPSEQ-conc [simp]:
shows $\text{SUPSEQ } (A \cdot B) = \text{SUPSEQ } A \cdot \text{SUPSEQ } B$
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 ≡ ⋃ (Atom ' UNIV)

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

lemma [simp]:
  shows (∪ a. {[a]})⋆ = UNIV
apply(auto)
apply(induct-tac x)
apply(auto)
apply(subgoal-tac [a] @ list ∈ (∪ 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 ⇒ '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 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 (λ y. 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 (λ w. Derivs w (lang r)))
proof −
  have ?thesis ←→ finite (UNIV // ≈ 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 ←→ 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 \( \lambda x. x \in B \Rightarrow y \in B \Rightarrow x \neq y \Rightarrow \exists w. \neg(x @ w \in A \leftrightarrow y @ w \in A) \)
  shows \( \neg \text{regular-lang} (A :: 'a :: finite list set) \)
  proof −
    have \( (\lambda w. \text{Derivs w A}) \cdot B \subseteq \text{range} (\lambda w. \text{Derivs w A}) \) by blast
    moreover from assms(2) have inj-on \( (\lambda w. \text{Derivs w A}) B \)
      by (auto simp: inj-on-def Derivs-def)
    with assms(1) have infinite \( (\lambda w. \text{Derivs w A}) \cdot B \)
      by (blast dest: finite-imageD)
    ultimately have 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 \( B \setminus x \in B \implies y \in B \implies x \neq y \implies \exists w. \neg(f x \circ w \in A \leftrightarrow f y \circ w \in A) \)

shows \( \neg\text{regular-lang } (A :: 'a :: \text{finite list set}) \)

proof (rule not-regular-langI)
  from assms(2) have inj-on \( f \) \( B \) by (force simp: inj-on-def)
  with \( \langle \text{infinite } B \rangle \) show \( \text{infinite } (f ' B) \) 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 \( \exists z \in A \) finite \( (\text{range } \delta) \) \( \text{length } z \geq n \)
  shows \( \exists u v w. z = u \circ v \circ w \land \text{length } (u \circ v) \leq n \land v \neq [] \land (\forall i. u \circ \text{repeat } i v \circ w \in A) \)

proof
  define \( P \) where \( P = \text{set } (\text{take } (\text{Suc } n) (\text{prefixes } z)) \)
  from \( \langle \text{length } z \geq n \rangle \) have \([\text{simp}]: \text{card } P = \text{Suc } n \)
    unfolding \( P \)-def by (subst distinct-card) (auto intro: distinct-take)
  have length-le: \( \text{length } y \leq n \) if \( y \in P \)
  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)
  also have \( \neg\text{inj-on } \delta \) \( P \) 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 \( \delta \)-def)
  from \( ab \) have prefix-ab: \( \text{prefix } a \circ \text{prefix } b \circ z \) by (auto simp: P-def dest: in-set-takeD)
  from \( ab \) have length-ab: \( \text{length } a \leq n \) \( \text{length } b \leq n \)
  by (simp-all add: length-le)
  have *: \( \exists \)thesis
    if \( uz': \text{prefix } u \circ z' \circ \text{prefix } z' \circ \text{length } z' \leq n \)
    \( u \neq z' \) \( \text{Derivs } z' A = \text{Derivs } u A \) for \( u z' \)
  proof
    from \( \langle \text{prefix } u \circ z' \rangle \) and \( \langle u \neq z' \rangle \)
    obtain \( v \) where \( v [\text{simp}]: z' = u \circ v v \neq [] \)
    by (auto simp: prefix-def)
    from \( \langle \text{prefix } z' \circ z \rangle \) obtain \( w \) where \( w [\text{simp}]: z = u \circ v \circ w \)
    by (auto simp: prefix-def)
  hence \([\text{simp}]: \text{Derivs } (\text{repeat } i v) \circ (\text{Derivs } u A) = \text{Derivs } u A \) 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
    λz. 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)
    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] guess n .
  from this[of z n] 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. \text{length} \ (\text{filter} \ \text{id} \ w) = \text{length} \ (\text{filter} \ \text{Not} \ w) \} \text{ (is \( \neg \)regular-lang \( ?A \))}

proof (rule not-regular-langI')

show \( \infty \) (UNIV :: nat set) by simp

next

fix \( m \) \( n :: \text{nat} \) assume \( m \neq n \)

hence \( \text{replicate} \ m \ \text{True} @ \text{replicate} \ m \ \text{False} \in \ ?A \) and

\( \text{replicate} \ n \ \text{True} @ \text{replicate} \ n \ \text{False} \notin \ ?A \) by simp-all

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

qed

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

lemma eq-replicate-iff:

\( xs = \text{replicate} \ n \ x \leftrightarrow \text{set} \ xs \subseteq \{ x \} \land \text{length} \ xs = n \)

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 \( \text{length} \ (u @ v) \leq n \) and \( v \neq [] \)

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

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

qed

lemma \( \neg \)regular-lang (range (\( \lambda i. \text{replicate} \ i \ \text{True} @ \text{replicate} \ i \ \text{False} \)) \text{ (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 \( \text{length-le}: \text{length} \ (u @ v) \leq n \) and \( v \neq [] \)

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

have \( w = w1 @ w2 \) by (simp add: w1-def w2-def)

have \( \text{replicate} \ n \ \text{True} = \text{take} \ n \ (\text{replicate} \ n \ \text{True} @ \text{replicate} \ n \ \text{False}) \) by simp

also note decomp

also have \( \text{take} \ n \ (u @ v @ w) = u @ v @ w1 \) using length-le by (simp add: w1-def)

finally have \( u @ v @ w1 = \text{replicate} \ n \ \text{True} \) by simp

then obtain \( i \ j \ k \)

44
where \( uvw1 \): \( n = i + j + k \) \( u = \text{replicate } i \text{ True} \) \( v = \text{replicate } j \text{ True} \) \( w1 = \text{replicate } k \text{ True} \)

by (elim replicate-eq-appendE) auto

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

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

end

References
