tphols-2011

By xingyuan

February 18, 2011

Contents

theory *Folds*
imports *Main*
begin

1 Folds for Sets

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

end

2 A general "while" combinator

theory *While-Combinator*
imports *Main*
begin

2.1 Partial version

```
definition while-option :: ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a option where
while-option b c s = (if (∃ k. ~ b ((c ^^ k) s))
  then Some ((c ^^ (LEAST k. ~ b ((c ^^ k) s))) s)
  else None)

theorem while-option-unfold[code]:
while-option b c s = (if b s then while-option b c (c s) else Some s)
proof cases
  assume b s
  show ?thesis
  proof (cases ∃ k. ~ b ((c ^^ k) s))
    case True
    then obtain k where 1: ~ b ((c ^^ k) s) ..
    with ‹b s› obtain l where k = Suc l by (cases k) auto
    with 1 have ~ b ((c ^^ l) (c s)) by (auto simp: funpow-swap1)
    then have 2: ∃ l. ~ b ((c ^^ l) (c s)) ..
    from 1
    have (LEAST k. ~ b ((c ^^ k) s)) = Suc (LEAST l. ~ b ((c ^^ Suc l) s))
      by (rule Least-Suc) (simp add: ‹b s›)
    also have ... = Suc (LEAST l. ~ b ((c ^^ l) (c s)))
      by (simp add: funpow-swap1)
    finally
    show ?thesis
      using True 2 ‹b s› by (simp add: funpow-swap1 while-option-def)
  next
    case False
    then have ~ (∃ l. ~ b ((c ^^ Suc l) s)) by blast
    then have ~ (∃ l. ~ b ((c ^^ l) (c s)))
      by (simp add: funpow-swap1)
    with False ‹b s› show ?thesis by (simp add: while-option-def)
  qed
next
  assume [simp]: ~ b s
  have least: (LEAST k. ~ b ((c ^^ k) s)) = 0
    by (rule Least-equality) auto
  moreover
  have ∃ k. ~ b ((c ^^ k) s) by (rule exI[of - 0::nat]) auto
  ultimately show ?thesis unfolding while-option-def by auto
qed

lemma while-option-stop:
assumes while-option b c s = Some t
shows ~ b t
proof -
  from assms have ex: ∃ k. ~ b ((c ^^ k) s)
  and t: t = (c ^^ (LEAST k. ~ b ((c ^^ k) s))) s
    by (auto simp: while-option-def split: if-splits)
  from LeastI-ex[OF ex]
```

```
  show ~ b t unfolding t .
qed

theorem while-option-rule:
assumes step: !!s. P s ==> b s ==> P (c s)
and result: while-option b c s = Some t
and init: P s
shows P t
proof -
  def k == LEAST k. ~ b ((c ^^ k) s)
  from assms have t: t = (c ^^ k) s
    by (simp add: while-option-def k-def split: if-splits)
  have 1: ALL i<k. b ((c ^^ i) s)
    by (auto simp: k-def dest: not-less-Least)

  { fix i assume i <= k then have P ((c ^^ i) s)
      by (induct i) (auto simp: init step 1) }
  thus P t by (auto simp: t)
qed
```

2.2 Total version

```
definition while :: ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a
where while b c s = the (while-option b c s)

lemma while-unfold:
  while b c s = (if b s then while b c (c s) else s)
unfolding while-def by (subst while-option-unfold) simp

lemma def-while-unfold:
  assumes fdef: f == while test do
  shows f x = (if test x then f(do x) else x)
unfolding fdef by (fact while-unfold)
```

The proof rule for *while*, where P is the invariant.

```
theorem while-rule-lemma:
  assumes invariant: !!s. P s ==> b s ==> P (c s)
    and terminate: !!s. P s ==> ¬ b s ==> Q s
    and wf: wf {(t, s). P s ∧ b s ∧ t = c s}
  shows P s ⟹ Q (while b c s)
  using wf
  apply (induct s)
  apply simp
  apply (subst while-unfold)
  apply (simp add: invariant terminate)
  done

theorem while-rule:
  [| P s;
```

```
      !!s. [| P s; b s  |] ==> P (c s);
      !!s. [| P s; ¬ b s  |] ==> Q s;
      wf r;
      !!s. [| P s; b s  |] ==> (c s, s) ∈ r |] ==>
   Q (while b c s)
  apply (rule while-rule-lemma)
     prefer 4 apply assumption
    apply blast
   apply blast
  apply (erule wf-subset)
  apply blast
  done


end

theory Myhill-1
imports Main Folds While-Combinator
begin
```

3 Preliminary definitions

types *lang* = *string set*

Sequential composition of two languages

definition
  Seq :: *lang* ⇒ *lang* ⇒ *lang* (**infixr** ;; *100*)
where
  A ;; $B = \{s_1 \ @ \ s_2 \mid s_1\ s_2.\ s_1 \in A \wedge s_2 \in B\}$

Some properties of operator ;;.

lemma *seq-add-left*:
  assumes *a*: $A = B$
  shows C ;; $A = C$;; B
using *a* **by** *simp*

lemma *seq-union-distrib-right*:
  shows $(A \cup B)$;; $C = (A$;; $C) \cup (B$;; $C)$
unfolding *Seq-def* **by** *auto*

lemma *seq-union-distrib-left*:
  shows C ;; $(A \cup B) = (C$;; $A) \cup (C$;; $B)$
unfolding *Seq-def* **by** *auto*

lemma *seq-intro*:
  assumes *a*: $x \in A$ $y \in B$
  shows $x \ @ \ y \in A$;; B
using *a* **by** (*auto simp*: *Seq-def*)

```
lemma seq-assoc:
  shows (A ;; B) ;; C = A ;; (B ;; C)
unfolding Seq-def
apply(auto)
apply(blast)
by (metis append-assoc)

lemma seq-empty [simp]:
  shows A ;; {[]} = A
  and   {[]} ;; A = A
by (simp-all add: Seq-def)
```

Power and Star of a language

```
fun
  pow :: lang ⇒ nat ⇒ lang (infixl ↑ 100)
where
  A ↑ 0 = {[]}
| A ↑ (Suc n) =  A ;; (A ↑ n)

definition
  Star :: lang ⇒ lang (-⋆ [101] 102)
where
  A⋆ ≡ (⋃n. A ↑ n)


lemma star-start[intro]:
  shows [] ∈ A⋆
proof −
  have [] ∈ A ↑ 0 by auto
  then show [] ∈ A⋆ unfolding Star-def by blast
qed

lemma star-step [intro]:
  assumes a: s1 ∈ A
  and     b: s2 ∈ A⋆
  shows s1 @ s2 ∈ A⋆
proof −
  from b obtain n where s2 ∈ A ↑ n unfolding Star-def by auto
  then have s1 @ s2 ∈ A ↑ (Suc n) using a by (auto simp add: Seq-def)
  then show s1 @ s2 ∈ A⋆ unfolding Star-def by blast
qed

lemma star-induct[consumes 1, case-names start step]:
  assumes a: x ∈ A⋆
  and     b: P []
  and     c: ⋀s1 s2. ⟦s1 ∈ A; s2 ∈ A⋆; P s2⟧ ⟹ P (s1 @ s2)
  shows P x
proof −
```

```
  from a obtain n where x ∈ A ↑ n unfolding Star-def by auto
  then show P x
    by (induct n arbitrary: x)
       (auto intro!: b c simp add: Seq-def Star-def)
qed

lemma star-intro1:
  assumes a: x ∈ A⋆
  and     b: y ∈ A⋆
  shows x @ y ∈ A⋆
using a b
by (induct rule: star-induct) (auto)

lemma star-intro2:
  assumes a: y ∈ A
  shows y ∈ A⋆
proof −
  from a have y @ [] ∈ A⋆ by blast
  then show y ∈ A⋆ by simp
qed

lemma star-intro3:
  assumes a: x ∈ A⋆
  and     b: y ∈ A
  shows x @ y ∈ A⋆
using a b by (blast intro: star-intro1 star-intro2)

lemma star-cases:
  shows A⋆ =  {[]} ∪ A ;; A⋆
proof
  { fix x
    have x ∈ A⋆ ⟹ x ∈ {[]} ∪ A ;; A⋆
      unfolding Seq-def
    by (induct rule: star-induct) (auto)
  }
  then show A⋆ ⊆ {[]} ∪ A ;; A⋆ by auto
next
  show {[]} ∪ A ;; A⋆ ⊆ A⋆
    unfolding Seq-def by auto
qed

lemma star-decom:
  assumes a: x ∈ A⋆ x ≠ []
  shows ∃ a b. x = a @ b ∧ a ≠ [] ∧ a ∈ A ∧ b ∈ A⋆
using a
by (induct rule: star-induct) (blast)+

lemma
  shows seq-Union-left:  B ;; (⋃n. A ↑ n) = (⋃n. B ;; (A ↑ n))
```

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

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

lemma *seq-star-comm*:
shows A ;; $A\star = A\star$;; A
unfolding *Star-def seq-Union-left*
unfolding *seq-pow-comm seq-Union-right*
by *simp*

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

lemma *pow-length*:
assumes *a*: $[] \notin A$
and *b*: $s \in A \uparrow Suc\ n$
shows $n < length\ s$
using *b*
proof (*induct n arbitrary*: *s*)
case *0*
have $s \in A \uparrow Suc\ 0$ **by** *fact*
with *a* **have** $s \neq []$ **by** *auto*
then show $0 < length\ s$ **by** *auto*
next
case (*Suc n*)
have *ih*: $\bigwedge s.\ s \in A \uparrow Suc\ n \Longrightarrow n < length\ s$ **by** *fact*
have $s \in A \uparrow Suc\ (Suc\ n)$ **by** *fact*
then obtain *s1 s2* **where** *eq*: $s = s1\ @\ s2$ **and** $*$: $s1 \in A$ **and** $**$: $s2 \in A \uparrow$
$Suc\ n$
by (*auto simp add*: *Seq-def*)
from *ih* $**$ **have** $n < length\ s2$ **by** *simp*
moreover have $0 < length\ s1$ **using** $*$ *a* **by** *auto*
ultimately show $Suc\ n < length\ s$ **unfolding** *eq*
by (*simp only*: *length-append*)
qed

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

4 A modified version of Arden's lemma

A helper lemma for Arden

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

theorem *arden*:
  assumes *nemp*: $[] \notin A$
  shows $X = X\ ;;\ A \cup B \longleftrightarrow X = B\ ;;\ A\star$
proof
  assume *eq*: $X = B\ ;;\ A\star$
  have $A\star = \{[]\} \cup A\star\ ;;\ A$
    unfolding *seq-star-comm*[*symmetric*]
    by (*rule star-cases*)
  then have $B\ ;;\ A\star = B\ ;;\ (\{[]\} \cup A\star\ ;;\ A)$
    by (*rule seq-add-left*)
  also have $\ldots = B \cup B\ ;;\ (A\star\ ;;\ A)$
    unfolding *seq-union-distrib-left* **by** *simp*
  also have $\ldots = B \cup (B\ ;;\ A\star)\ ;;\ A$
    by (*simp only*: *seq-assoc*)
  finally show $X = X\ ;;\ A \cup B$
    using *eq* **by** *blast*
next
  assume *eq*: $X = X\ ;;\ A \cup B$
  { fix *n::nat*
    have $B\ ;;\ (A \uparrow n) \subseteq X$ **using** *arden-helper*[*OF eq*, *of n*] **by** *auto* **}**
  then have $B\ ;;\ A\star \subseteq X$
    unfolding *Seq-def Star-def UNION-def* **by** *auto*
  moreover
  { fix *s::string*
    obtain *k* **where** $k = \mathit{length}\ s$ **by** *auto*
    then have *not-in*: $s \notin X\ ;;\ (A \uparrow \mathit{Suc}\ k)$

```
    using seq-pow-length[OF nemp] by blast
  assume s ∈ X
  then have s ∈ X ;; (A ↑ Suc k) ∪ (⋃m∈{0..k}. B ;; (A ↑ m))
    using arden-helper[OF eq, of k] by auto
  then have s ∈ (⋃m∈{0..k}. B ;; (A ↑ m)) using not-in by auto
  moreover
  have (⋃m∈{0..k}. B ;; (A ↑ m)) ⊆ (⋃n. B ;; (A ↑ n)) by auto
  ultimately
  have s ∈ B ;; A⋆
    unfolding seq-Union-left Star-def by auto }
 then have X ⊆ B ;; A⋆ by auto
 ultimately
 show X = B ;; A⋆ by simp
qed
```

5 Regular Expressions

```
datatype rexp =
  NULL
| EMPTY
| CHAR char
| SEQ rexp rexp
| ALT rexp rexp
| STAR rexp
```

The function L is overloaded, with the idea that $L\ x$ evaluates to the language represented by the object x.

```
consts L:: 'a ⇒ lang

overloading L-rexp ≡ L::  rexp ⇒ lang
begin
fun
  L-rexp :: rexp ⇒ lang
where
    L-rexp (NULL) = {}
  | L-rexp (EMPTY) = {[]}
  | L-rexp (CHAR c) = {[c]}
  | L-rexp (SEQ r1 r2) = (L-rexp r1) ;; (L-rexp r2)
  | L-rexp (ALT r1 r2) = (L-rexp r1) ∪ (L-rexp r2)
  | L-rexp (STAR r) = (L-rexp r)⋆
end
```

ALT-combination of a set or regulare expressions

```
abbreviation
  Setalt  (⨄- [1000] 999)
where
  ⨄A ≡ folds ALT NULL A
```

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

lemma *folds-alt-simp* [*simp*]:
  fixes *rs*::*rexp set*
  assumes *a*: *finite rs*
  shows $L\ (\biguplus rs) = \bigcup\ (L\ \text{`}\ rs)$
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

6 Direction *finite partition* ⇒ *regular language*

Just a technical lemma for collections and pairs

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

Myhill-Nerode relation

definition
  str-eq-rel :: *lang* ⇒ (*string* × *string*) *set* (≈- [*100*] *100*)
where
  $\approx A \equiv \{(x,\ y).\ \ (\forall z.\ x\ @\ z \in A \longleftrightarrow y\ @\ z \in A)\}$

Among the equivalence clases of $\approx A$, the set *finals* A singles out those which contains the strings from A.

definition
  finals :: *lang* ⇒ *lang set*
where
  finals $A \equiv \{\approx A\ \text{``}\ \{s\}\ |\ s\ .\ s \in A\}$

lemma *lang-is-union-of-finals*:
  shows $A = \bigcup$ *finals* A
unfolding *finals-def*
unfolding *Image-def*
unfolding *str-eq-rel-def*
apply(*auto*)
apply(*drule-tac* $x = []$ **in** *spec*)
apply(*auto*)
done

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

7 Equational systems

The two kinds of terms in the rhs of equations.

datatype *rhs-item* =
  Lam rexp
| *Trn lang rexp*

overloading *L-rhs-item* ≡ *L*:: *rhs-item* ⇒ *lang*
begin
  fun *L-rhs-item*:: *rhs-item* ⇒ *lang*
  where
    L-rhs-item (*Lam r*) = *L r*
  | *L-rhs-item* (*Trn X r*) = *X* ;; *L r*
end

overloading *L-rhs* ≡ *L*:: *rhs-item set* ⇒ *lang*
begin
   fun *L-rhs*:: *rhs-item set* ⇒ *lang*
   where
     L-rhs rhs = ⋃ (*L* ‘ *rhs*)
end

lemma *L-rhs-union-distrib*:
  fixes *A B*::*rhs-item set*
  shows *L A* ∪ *L B* = *L* (*A* ∪ *B*)
by *simp*

Transitions between equivalence classes

definition
  transition :: *lang* ⇒ *char* ⇒ *lang* ⇒ *bool* (- ⊨-⇒- [*100*,*100*,*100*] *100*)
where
  Y ⊨*c*⇒ *X* ≡ *Y* ;; {[*c*]} ⊆ *X*

Initial equational system

definition
  Init-rhs CS X ≡
    if ([] ∈ *X*) *then*
      {*Lam EMPTY*} ∪ {*Trn Y* (*CHAR c*) | *Y c*. *Y* ∈ *CS* ∧ *Y* ⊨*c*⇒ *X*}
    else
      {*Trn Y* (*CHAR c*)| *Y c*. *Y* ∈ *CS* ∧ *Y* ⊨*c*⇒ *X*}

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

8 Arden Operation on equations

The function *attach-rexp r item* SEQ-composes *r* to the right of every rhs-item.

fun
  append-rexp :: *rexp* ⇒ *rhs-item* ⇒ *rhs-item*
where
  append-rexp r (*Lam rexp*)   = *Lam* (*SEQ rexp r*)
| *append-rexp r* (*Trn X rexp*) = *Trn X* (*SEQ rexp r*)

definition
  append-rhs-rexp rhs rexp ≡ (*append-rexp rexp*) ' *rhs*

definition
  Arden X rhs ≡
    append-rhs-rexp (*rhs* − {*Trn X r* | *r*. *Trn X r* ∈ *rhs*}) (*STAR* (⨄ {*r*. *Trn X r* ∈ *rhs*}))

9 Substitution Operation on equations

Suppose and equation $X = xrhs$, *Subst* substitutes all occurences of X in *rhs* by *xrhs*.

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

eqs-subst ES X xrhs substitutes *xrhs* into every equation of the equational system *ES*.

types *esystem* = (*lang* × *rhs-item set*) *set*

definition
  Subst-all :: *esystem* ⇒ *lang* ⇒ *rhs-item set* ⇒ *esystem*
where
  Subst-all ES X xrhs ≡ {(*Y*, *Subst yrhs X xrhs*) | *Y yrhs*. (*Y*, *yrhs*) ∈ *ES*}

The following term *remove ES Y yrhs* removes the equation $Y = yrhs$ from equational system *ES* by replacing all occurences of Y by its definition (using *eqs-subst*). The Y-definition is made non-recursive using Arden's transformation *arden-variate Y yrhs*.

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

10 While-combinator

The following term *Iter X ES* represents one iteration in the while loop. It arbitrarily chooses a Y different from X to remove.

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

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*)

The following term *Reduce X ES* repeatedly removes characteriztion equations for unknowns other than X until one is left.

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

definition
$Solve\ X\ ES \equiv while\ Cond\ (Iter\ X)\ ES$

Since the *while* combinator from HOL library is used to implement *Solve X ES*, the induction principle *while-rule* is used to proved the desired properties of *Solve X ES*. For this purpose, an invariant predicate *invariant* is defined in terms of a series of auxilliary predicates:

11 Invariants

Every variable is defined at most once in *ES*.

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

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

definition
$sound\text{-}eqs\ ES \equiv \forall (X,\ rhs) \in ES.\ X = L\ rhs$

ardenable rhs requires regular expressions occuring in transitional items of *rhs* do not contain empty string. This is necessary for the application of Arden's transformation to *rhs*.

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

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

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

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

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

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

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

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

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

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

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

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

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

definition
invariant ES $\equiv$ *finite ES*
$\wedge$ *finite-rhs ES*
$\wedge$ *sound-eqs ES*
$\wedge$ *distinct-equas ES*
$\wedge$ *ardenable-all ES*
$\wedge$ *valid-eqs ES*

lemma *invariantI*:
assumes *sound-eqs ES finite ES distinct-equas ES ardenable-all ES*
finite-rhs ES valid-eqs ES
shows *invariant ES*
using *assms* **by** (*simp add*: *invariant-def*)

11.1 The proof of this direction

11.1.1 Basic properties

The following are some basic properties of the above definitions.

lemma *finite-Trn*:
  assumes *fin*: *finite rhs*
  shows *finite* {*r*. *Trn Y r* ∈ *rhs*}
proof −
  have *finite* {*Trn Y r* | *Y r*. *Trn Y r* ∈ *rhs*}
    by (*rule rev-finite-subset*[*OF fin*]) (*auto*)
  then have *finite* ((λ(*Y*, *r*). *Trn Y r*) ‘ {(*Y*, *r*) | *Y r*. *Trn Y r* ∈ *rhs*})
    by (*simp add*: *image-Collect*)
  then have *finite* {(*Y*, *r*) | *Y r*. *Trn Y r* ∈ *rhs*}
    by (*erule-tac finite-imageD*) (*simp add*: *inj-on-def*)
  then show *finite* {*r*. *Trn Y r* ∈ *rhs*}
    by (*erule-tac f=snd* **in** *finite-surj*) (*auto simp add*: *image-def*)
qed

lemma *finite-Lam*:
  assumes *fin*: *finite rhs*
  shows *finite* {*r*. *Lam r* ∈ *rhs*}
proof −
  have *finite* {*Lam r* | *r*. *Lam r* ∈ *rhs*}
    by (*rule rev-finite-subset*[*OF fin*]) (*auto*)
  then show *finite* {*r*. *Lam r* ∈ *rhs*}
    apply(*simp add*: *image-Collect*[*symmetric*])
    apply(*erule finite-imageD*)
    apply(*auto simp add*: *inj-on-def*)
    done
qed

lemma *rexp-of-empty*:
  assumes *finite*: *finite rhs*
  and *nonempty*: *ardenable rhs*
  shows [] ∉ *L* (⨄ {*r*. *Trn X r* ∈ *rhs*})
using *finite nonempty ardenable-def*
using *finite-Trn*[*OF finite*]
by *auto*

lemma *lang-of-rexp-of*:
  assumes *finite*:*finite rhs*
  shows *L* ({*Trn X r*| *r*. *Trn X r* ∈ *rhs*}) = *X* ;; (*L* (⨄{*r*. *Trn X r* ∈ *rhs*}))
proof −
  have *finite* {*r*. *Trn X r* ∈ *rhs*}
    by (*rule finite-Trn*[*OF finite*])
  then show *?thesis*
    apply(*auto simp add*: *Seq-def*)
    apply(*rule-tac* $x = s_1$ **in** *exI*, *rule-tac* $x = s_2$ **in** *exI*)
    apply(*auto*)
    apply(*rule-tac x*= *Trn X xa* **in** *exI*)
    apply(*auto simp add*: *Seq-def*)
    done
qed

lemma *lang-of-append*:
  L (*append-rexp r rhs-item*) = L *rhs-item* ;; L r
by (*induct rule*: *append-rexp.induct*)
  (*auto simp add*: *seq-assoc*)

lemma *lang-of-append-rhs*:
  L (*append-rhs-rexp rhs r*) = L *rhs* ;; L r
unfolding *append-rhs-rexp-def*
by (*auto simp add*: *Seq-def lang-of-append*)

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*)

11.1.2 Intialization

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

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

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

```
qed

lemma l-eq-r-in-eqs:
  assumes X-in-eqs: (X, rhs) ∈ Init (UNIV // ≈A)
  shows X = L rhs
proof
  show X ⊆ L rhs
  proof
    fix x
    assume (1): x ∈ X
    show x ∈ L rhs
    proof (cases x = [])
      assume empty: x = []
      thus ?thesis using X-in-eqs (1)
        by (auto simp: Init-def Init-rhs-def)
    next
      assume not-empty: x ≠ []
      then obtain clist c where decom: x = clist @ [c]
        by (case-tac x rule:rev-cases, auto)
      have X ∈ UNIV // ≈A using X-in-eqs by (auto simp:Init-def)
      then obtain Y
        where Y ∈ UNIV // ≈A
        and Y ;; {[c]} ⊆ X
        and clist ∈ Y
        using decom (1) every-eqclass-has-transition by blast
      hence
        x ∈ L {Trn Y (CHAR c)| Y c. Y ∈ UNIV // ≈A ∧ Y ⊨c⇒ X}
        unfolding transition-def
        using (1) decom
        by (simp, rule-tac x = Trn Y (CHAR c) in exI, simp add:Seq-def)
      thus ?thesis using X-in-eqs (1)
        by (simp add: Init-def Init-rhs-def)
    qed
  qed
next
  show L rhs ⊆ X using X-in-eqs
    by (auto simp:Init-def Init-rhs-def transition-def)
qed

lemma test:
  assumes X-in-eqs: (X, rhs) ∈ Init (UNIV // ≈A)
  shows X = ⋃ (L ‘ rhs)
using assms
by (drule-tac l-eq-r-in-eqs) (simp)


lemma finite-Init-rhs:
  assumes finite: finite CS
  shows finite (Init-rhs CS X)
```

```
proof−
  def S ≡ {(Y, c)| Y c. Y ∈ CS ∧ Y ;; {[c]} ⊆ X}
  def h ≡ λ (Y, c). Trn Y (CHAR c)
  have finite (CS × (UNIV::char set)) using finite by auto
  then have finite S using S-def
    by (rule-tac B = CS × UNIV in finite-subset) (auto)
  moreover have {Trn Y (CHAR c) |Y c. Y ∈ CS ∧ Y ;; {[c]} ⊆ X} = h ‘ S
    unfolding S-def h-def image-def by auto
  ultimately
  have finite {Trn Y (CHAR c) |Y c. Y ∈ CS ∧ Y ;; {[c]} ⊆ X} by auto
 then show finite (Init-rhs CS X) unfolding Init-rhs-def transition-def by simp
qed

lemma Init-ES-satisfies-invariant:
  assumes finite-CS: finite (UNIV // ≈A)
  shows invariant (Init (UNIV // ≈A))
proof (rule invariantI)
  show sound-eqs (Init (UNIV // ≈A))
    unfolding sound-eqs-def
    using l-eq-r-in-eqs by auto
  show finite (Init (UNIV // ≈A)) using finite-CS
    unfolding Init-def by simp
  show distinct-equas (Init (UNIV // ≈A))
    unfolding distinct-equas-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 valid-eqs (Init (UNIV // ≈A))
    unfolding valid-eqs-def Init-def Init-rhs-def rhss-def lhss-def
    by auto
qed
```

11.1.3 Interation step

From this point until *iteration-step*, the correctness of the iteration step *Iter X ES* is proved.

```
lemma Arden-keeps-eq:
  assumes l-eq-r: X = L rhs
  and not-empty: ardenable rhs
  and finite: finite rhs
  shows X = L (Arden X rhs)
proof −
  def A ≡ L (⨄{r. Trn X r ∈ rhs})
  def b ≡ rhs − {Trn X r | r. Trn X r ∈ rhs}
  def B ≡ L b
  have X = B ;; A⋆
```

```
  proof -
    have L rhs = L({Trn X r | r. Trn X r ∈ rhs} ∪ b) by (auto simp: b-def)
    also have ... = X ;; A ∪ B
      unfolding L-rhs-union-distrib[symmetric]
      by (simp only: lang-of-rexp-of finite B-def A-def)
    finally show ?thesis
      apply(rule-tac arden[THEN iffD1])
      apply(simp (no-asm) add: A-def)
      using finite-Trn[OF finite] not-empty
      apply(simp add: ardenable-def)
      using l-eq-r
      apply(simp)
      done
  qed
  moreover have L (Arden X rhs) = B ;; A⋆
    by (simp only:Arden-def L-rhs-union-distrib lang-of-append-rhs
                  B-def A-def b-def L-rexp.simps seq-union-distrib-left)
   ultimately show ?thesis by simp
qed

lemma append-keeps-finite:
  finite rhs ⟹ finite (append-rhs-rexp rhs r)
by (auto simp:append-rhs-rexp-def)

lemma Arden-keeps-finite:
  finite rhs ⟹ finite (Arden X rhs)
by (auto simp:Arden-def append-keeps-finite)

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

lemma nonempty-set-sub:
  ardenable rhs ⟹ ardenable (rhs - A)
by (auto simp:ardenable-def)

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

lemma Arden-keeps-nonempty:
  ardenable rhs ⟹ ardenable (Arden X rhs)
by (simp only:Arden-def append-keeps-nonempty nonempty-set-sub)


lemma Subst-keeps-nonempty:
  ⟦ardenable rhs; ardenable xrhs⟧ ⟹ ardenable (Subst rhs X xrhs)
by (simp only:Subst-def append-keeps-nonempty  nonempty-set-union nonempty-set-sub)
```

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

lemma *Subst-keeps-finite-rhs*:
  ⟦*finite rhs*; *finite yrhs*⟧ ⟹ *finite* (*Subst rhs Y yrhs*)
by (*auto simp*:*Subst-def append-keeps-finite*)

lemma *Subst-all-keeps-finite*:
  assumes *finite*: *finite ES*
  shows *finite* (*Subst-all ES Y yrhs*)
proof −
  def $eqns \equiv \{(X{::}lang,\ rhs) \mid X\ rhs.\ (X,\ rhs) \in ES\}$
  def $h \equiv \lambda(X{::}lang,\ rhs).\ (X,\ Subst\ rhs\ Y\ yrhs)$
  have *finite* (*h* ‘ *eqns*) **using** *finite h-def eqns-def* **by** *auto*
  moreover
  have *Subst-all ES Y yrhs* = *h* ‘ *eqns* **unfolding** *h-def eqns-def Subst-all-def* **by** *auto*
  ultimately
  show *finite* (*Subst-all ES Y yrhs*) **by** *simp*
qed

lemma *Subst-all-keeps-finite-rhs*:
  ⟦*finite-rhs ES*; *finite yrhs*⟧ ⟹ *finite-rhs* (*Subst-all ES Y yrhs*)
by (*auto intro*:*Subst-keeps-finite-rhs simp add*:*Subst-all-def finite-rhs-def*)

lemma *append-rhs-keeps-cls*:

```
  rhss (append-rhs-rexp rhs r) = rhss rhs
apply (auto simp:rhss-def append-rhs-rexp-def)
apply (case-tac xa, auto simp:image-def)
by (rule-tac x = SEQ ra r in exI, rule-tac x = Trn x ra in bexI, simp+)

lemma Arden-removes-cl:
  rhss (Arden Y yrhs) = rhss yrhs − {Y}
apply (simp add:Arden-def append-rhs-keeps-cls)
by (auto simp:rhss-def)

lemma lhss-keeps-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-keeps-cls rhss-union-distrib)
by (auto simp:rhss-def)

lemma Subst-all-keeps-valid-eqs:
  assumes sc: valid-eqs (ES ∪ {(Y, yrhs)})        (is valid-eqs ?A)
  shows valid-eqs (Subst-all ES Y (Arden Y yrhs))  (is valid-eqs ?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 simp
          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:valid-eqs-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 valid-eqs-def lhss-def)
        ultimately show ?thesis by auto
      qed
      ultimately show ?thesis by simp
```

```
    qed
  } thus ?thesis by (auto simp only:Subst-all-def valid-eqs-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 = L yrhs
    using invariant-ES by (simp only:invariant-def sound-eqs-def, blast)
   have finite-yrhs: finite yrhs
    using invariant-ES by (auto simp:invariant-def finite-rhs-def)
  have nonempty-yrhs: ardenable yrhs
    using invariant-ES by (auto simp:invariant-def ardenable-all-def)
  show sound-eqs (Subst-all ES Y (Arden Y yrhs))
  proof -
    have Y = L (Arden Y yrhs)
      using Y-eq-yrhs invariant-ES finite-yrhs
      using finite-Trn[OF finite-yrhs]
      apply(rule-tac Arden-keeps-eq)
      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 sound-eqs-def Subst-all-def
      by (auto simp add: Subst-keeps-eq simp del: L-rhs.simps)
  qed
  show finite (Subst-all ES Y (Arden Y yrhs))
    using invariant-ES by (simp add:invariant-def Subst-all-keeps-finite)
  show distinct-equas (Subst-all ES Y (Arden Y yrhs))
    using invariant-ES
    unfolding distinct-equas-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 prems invariant-ES
        by (auto simp add:invariant-def ardenable-all-def)
      with nonempty-yrhs
      have ardenable (Subst rhs Y (Arden Y yrhs))
        by (simp add:nonempty-yrhs
               Subst-keeps-nonempty Arden-keeps-nonempty)
    } 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-keeps-finite by simp
    qed
    ultimately show ?thesis
      by (simp add:Subst-all-keeps-finite-rhs)
  qed
  show valid-eqs (Subst-all ES Y (Arden Y yrhs))
    using invariant-ES Subst-all-keeps-valid-eqs by (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 -
  def f ≡ λ x. ((fst x)::lang, Subst (snd x) X (Arden X rhs))
  def 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 distinct-equas-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 distinct-equas-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*
      thm *Subst-all-satisfies-invariant*
      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-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 distinct-equas-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
```

11.1.4 Conclusion of the proof

```
lemma Solve:
  assumes fin: finite (UNIV // ≈A)
  and     X-in: X ∈ (UNIV // ≈A)
  shows ∃ rhs. Solve X (Init (UNIV // ≈A)) = {(X, rhs)} ∧ invariant {(X, rhs)}
proof −
  def Inv ≡ λES. invariant ES ∧ (∃ rhs. (X, rhs) ∈ 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:
  assumes finite-CS: finite (UNIV // ≈A)
  and X-in-CS: X ∈ (UNIV // ≈A)
  shows ∃ r::rexp. X = L r
proof −
  from finite-CS X-in-CS
  obtain xrhs where Inv-ES: invariant {(X, xrhs)}
    using Solve by metis

  def A ≡ Arden X xrhs
  have rhss xrhs ⊆ {X} using Inv-ES
    unfolding valid-eqs-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-keeps-finite by auto
  then have fin: finite {r. Lam r ∈ A} by (rule finite-Lam)

  have X = L xrhs using Inv-ES unfolding invariant-def sound-eqs-def
    by simp
  then have X = L A using Inv-ES
    unfolding A-def invariant-def ardenable-all-def finite-rhs-def
    by (rule-tac Arden-keeps-eq) (simp-all add: finite-Trn)
  then have X = L {Lam r | r. Lam r ∈ A} using eq by simp
  then have X = L (⨄{r. Lam r ∈ A}) using fin by auto
  then show ∃ r::rexp. X = L r by blast
qed

lemma bchoice-finite-set:
  assumes a: ∀ x ∈ S. ∃ y. x = f y
  and     b: finite S
```

shows $\exists$ *ys*. ($\bigcup$ *S*) = $\bigcup$(*f* ' *ys*) $\wedge$ *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*:
assumes *finite-CS*: *finite* (*UNIV* // $\approx$*A*)
shows $\exists$ *r*::*rexp*. *A* = *L r*
proof −
have *fin*: *finite* (*finals A*)
using *finals-in-partitions finite-CS* **by** (*rule finite-subset*)
have $\forall$ *X* $\in$ (*UNIV* // $\approx$*A*). $\exists$ *r*::*rexp*. *X* = *L r*
using *finite-CS every-eqcl-has-reg* **by** *blast*
then have *a*: $\forall$ *X* $\in$ *finals A*. $\exists$ *r*::*rexp*. *X* = *L r*
using *finals-in-partitions* **by** *auto*
then obtain *rs*::*rexp set* **where** $\bigcup$ (*finals A*) = $\bigcup$(*L* ' *rs*) *finite rs*
using *fin* **by** (*auto dest*: *bchoice-finite-set*)
then have *A* = *L* ($\biguplus$*rs*)
unfolding *lang-is-union-of-finals*[*symmetric*] **by** *simp*
then show $\exists$ *r*::*rexp*. *A* = *L r* **by** *blast*
qed

end

12 List prefixes and postfixes

theory *List-Prefix*
imports *List Main*
begin

12.1 Prefix order on lists

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

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

definition
strict-prefix-def: *xs* < *ys* $\longleftrightarrow$ *xs* $\leq$ *ys* $\wedge$ *xs* $\neq$ (*ys*::'*a list*)

definition
bot = []

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

```
end

lemma prefixI [intro?]: ys = xs @ zs ==> xs ≤ ys
  unfolding prefix-def by blast

lemma prefixE [elim?]:
  assumes xs ≤ ys
  obtains zs where ys = xs @ zs
  using assms unfolding prefix-def by blast

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

lemma strict-prefixE' [elim?]:
  assumes xs < ys
  obtains z zs where ys = xs @ z # zs
proof −
  from ‹xs < ys› obtain us where ys = xs @ us and xs ≠ ys
    unfolding strict-prefix-def prefix-def by blast
  with that show ?thesis by (auto simp add: neq-Nil-conv)
qed

lemma strict-prefixI [intro?]: xs ≤ ys ==> xs ≠ ys ==> xs < (ys::'a list)
  unfolding strict-prefix-def by blast

lemma strict-prefixE [elim?]:
  fixes xs ys :: 'a list
  assumes xs < ys
  obtains xs ≤ ys and xs ≠ ys
  using assms unfolding strict-prefix-def by blast
```

12.2 Basic properties of prefixes

```
theorem Nil-prefix [iff]: [] ≤ xs
  by (simp add: prefix-def)

theorem prefix-Nil [simp]: (xs ≤ []) = (xs = [])
  by (induct xs) (simp-all add: prefix-def)

lemma prefix-snoc [simp]: (xs ≤ ys @ [y]) = (xs = ys @ [y] ∨ xs ≤ ys)
proof
  assume xs ≤ ys @ [y]
  then obtain zs where zs: ys @ [y] = xs @ zs ..
  show xs = ys @ [y] ∨ xs ≤ ys
    by (metis append-Nil2 butlast-append butlast-snoc prefixI zs)
next
  assume xs = ys @ [y] ∨ xs ≤ ys
  then show xs ≤ ys @ [y]
```

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

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

lemma *less-eq-list-code* [*code*]:
$([]::'a::\{equal, ord\}\ list) \leq xs \longleftrightarrow True$
$(x::'a::\{equal, ord\}) \# xs \leq [] \longleftrightarrow False$
$(x::'a::\{equal, ord\}) \# xs \leq y \# ys \longleftrightarrow x = y \wedge xs \leq ys$
by *simp-all*

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

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

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

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

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

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

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

theorem *prefix-length-le*: $xs \leq ys ==> length\ xs \leq length\ ys$
by (*auto simp add*: *prefix-def*)

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

lemma *set-mono-prefix*: $xs \leq ys \Longrightarrow set\ xs \subseteq set\ ys$

```
  by (auto simp add: prefix-def)

lemma take-is-prefix: take n xs ≤ xs
  unfolding prefix-def by (metis append-take-drop-id)

lemma map-prefixI: xs ≤ ys ⟹ map f xs ≤ map f ys
  by (auto simp: prefix-def)

lemma prefix-length-less: xs < ys ⟹ length xs < length ys
  by (auto simp: strict-prefix-def prefix-def)

lemma strict-prefix-simps [simp, code]:
  xs < [] ⟷ False
  [] < x # xs ⟷ True
  x # xs < y # ys ⟷ x = y ∧ xs < ys
  by (simp-all add: strict-prefix-def cong: conj-cong)

lemma take-strict-prefix: xs < ys ⟹ take n xs < ys
  apply (induct n arbitrary: xs ys)
   apply (case-tac ys, simp-all)[1]
  apply (metis order-less-trans strict-prefixI take-is-prefix)
  done

lemma not-prefix-cases:
  assumes pfx: ¬ ps ≤ ls
  obtains
    (c1) ps ≠ [] and ls = []
  | (c2) a as x xs where ps = a#as and ls = x#xs and x = a and ¬ as ≤ xs
  | (c3) a as x xs where ps = a#as and ls = x#xs and x ≠ a
proof (cases ps)
  case Nil then show ?thesis using pfx by simp
next
  case (Cons a as)
  note c = ‹ps = a#as›
  show ?thesis
  proof (cases ls)
    case Nil then show ?thesis by (metis append-Nil2 pfx c1 same-prefix-nil)
  next
    case (Cons x xs)
    show ?thesis
    proof (cases x = a)
      case True
      have ¬ as ≤ xs using pfx c Cons True by simp
      with c Cons True show ?thesis by (rule c2)
    next
      case False
      with c Cons show ?thesis by (rule c3)
    qed
  qed
```

```
qed

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

12.3 Parallel lists

```
definition
  parallel :: 'a list => 'a list => bool  (infixl ∥ 50) where
  (xs ∥ ys) = (¬ xs ≤ ys ∧ ¬ ys ≤ xs)

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

lemma parallelE [elim]:
  assumes xs ∥ ys
  obtains ¬ xs ≤ ys ∧ ¬ ys ≤ xs
  using assms unfolding parallel-def by blast

theorem prefix-cases:
  obtains xs ≤ ys | ys < xs | xs ∥ ys
  unfolding parallel-def strict-prefix-def by blast

theorem parallel-decomp:
  xs ∥ ys ==> ∃ as b bs c cs. b ≠ c ∧ xs = as @ b # bs ∧ ys = as @ c # cs
proof (induct xs rule: rev-induct)
  case Nil
  then have False by auto
  then show ?case ..
next
  case (snoc x xs)
  show ?case
  proof (rule prefix-cases)
    assume le: xs ≤ ys
    then obtain ys' where ys: ys = xs @ ys' ..
```

```
    show ?thesis
    proof (cases ys′)
      assume ys′ = []
      then show ?thesis by (metis append-Nil2 parallelE prefixI snoc.prems ys)
    next
      fix c cs assume ys′: ys′ = c # cs
      then show ?thesis
        by (metis Cons-eq-appendI eq-Nil-appendI parallelE prefixI
          same-prefix-prefix snoc.prems ys)
    qed
  next
    assume ys < xs then have ys ≤ xs @ [x] by (simp add: strict-prefix-def)
    with snoc have False by blast
    then show ?thesis ..
  next
    assume xs ‖ ys
    with snoc obtain as b bs c cs where neq: (b::′a) ≠ c
      and xs: xs = as @ b # bs and ys: ys = as @ c # cs
      by blast
    from xs have xs @ [x] = as @ b # (bs @ [x]) by simp
    with neq ys show ?thesis by blast
  qed
qed

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

lemma parallel-appendI: xs ‖ ys ⟹ x = xs @ xs′ ⟹ y = ys @ ys′ ⟹ x ‖ y
  by (simp add: parallel-append)

lemma parallel-commute: a ‖ b ⟷ b ‖ a
  unfolding parallel-def by auto
```

12.4 Postfix order on lists

```
definition
  postfix :: ′a list => ′a list => bool  ((-/ >>= -) [51, 50] 50) where
  (xs >>= ys) = (∃ zs. xs = zs @ ys)

lemma postfixI [intro?]: xs = zs @ ys ==> xs >>= ys
  unfolding postfix-def by blast

lemma postfixE [elim?]:
  assumes xs >>= ys
  obtains zs where xs = zs @ ys
  using assms unfolding postfix-def by blast
```

```
lemma postfix-refl [iff]: xs >>= xs
  by (auto simp add: postfix-def)
lemma postfix-trans: ⟦xs >>= ys; ys >>= zs⟧ ⟹ xs >>= zs
  by (auto simp add: postfix-def)
lemma postfix-antisym: ⟦xs >>= ys; ys >>= xs⟧ ⟹ xs = ys
  by (auto simp add: postfix-def)

lemma Nil-postfix [iff]: xs >>= []
  by (simp add: postfix-def)
lemma postfix-Nil [simp]: ([] >>= xs) = (xs = [])
  by (auto simp add: postfix-def)

lemma postfix-ConsI: xs >>= ys ⟹ x#xs >>= ys
  by (auto simp add: postfix-def)
lemma postfix-ConsD: xs >>= y#ys ⟹ xs >>= ys
  by (auto simp add: postfix-def)

lemma postfix-appendI: xs >>= ys ⟹ zs @ xs >>= ys
  by (auto simp add: postfix-def)
lemma postfix-appendD: xs >>= zs @ ys ⟹ xs >>= ys
  by (auto simp add: postfix-def)

lemma postfix-is-subset: xs >>= ys ==> set ys ⊆ set xs
proof −
  assume xs >>= ys
  then obtain zs where xs = zs @ ys ..
  then show ?thesis by (induct zs) auto
qed

lemma postfix-ConsD2: x#xs >>= y#ys ==> xs >>= ys
proof −
  assume x#xs >>= y#ys
  then obtain zs where x#xs = zs @ y#ys ..
  then show ?thesis
    by (induct zs) (auto intro!: postfix-appendI postfix-ConsI)
qed

lemma postfix-to-prefix [code]: xs >>= ys ⟷ rev ys ≤ rev xs
proof
  assume xs >>= ys
  then obtain zs where xs = zs @ ys ..
  then have rev xs = rev ys @ rev zs by simp
  then show rev ys <= rev xs ..
next
  assume rev ys <= rev xs
  then obtain zs where rev xs = rev ys @ zs ..
  then have rev (rev xs) = rev zs @ rev (rev ys) by simp
  then have xs = rev zs @ ys by simp
```

then show *xs* >>= *ys* ..
qed

lemma *distinct-postfix*: *distinct xs* ⟹ *xs* >>= *ys* ⟹ *distinct ys*
by (*clarsimp elim*!: *postfixE*)

lemma *postfix-map*: *xs* >>= *ys* ⟹ *map f xs* >>= *map f ys*
by (*auto elim*!: *postfixE intro*: *postfixI*)

lemma *postfix-drop*: *as* >>= *drop n as*
unfolding *postfix-def*
apply (*rule exI* [**where** *x* = *take n as*])
apply *simp*
done

lemma *postfix-take*: *xs* >>= *ys* ⟹ *xs* = *take* (*length xs* − *length ys*) *xs* @ *ys*
by (*clarsimp elim*!: *postfixE*)

lemma *parallelD1*: *x* ∥ *y* ⟹ ¬ *x* ≤ *y*
by *blast*

lemma *parallelD2*: *x* ∥ *y* ⟹ ¬ *y* ≤ *x*
by *blast*

lemma *parallel-Nil1* [*simp*]: ¬ *x* ∥ []
unfolding *parallel-def* **by** *simp*

lemma *parallel-Nil2* [*simp*]: ¬ [] ∥ *x*
unfolding *parallel-def* **by** *simp*

lemma *Cons-parallelI1*: *a* ≠ *b* ⟹ *a* # *as* ∥ *b* # *bs*
by *auto*

lemma *Cons-parallelI2*: ⟦ *a* = *b*; *as* ∥ *bs* ⟧ ⟹ *a* # *as* ∥ *b* # *bs*
by (*metis Cons-prefix-Cons parallelE parallelI*)

lemma *not-equal-is-parallel*:
assumes *neq*: *xs* ≠ *ys*
and *len*: *length xs* = *length ys*
shows *xs* ∥ *ys*
using *len neq*
proof (*induct rule*: *list-induct2*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a as b bs*)
have *ih*: *as* ≠ *bs* ⟹ *as* ∥ *bs* **by** *fact*
show *?case*
proof (*cases a* = *b*)

```
    case True
    then have as ≠ bs using Cons by simp
    then show ?thesis by (rule Cons-parallelI2 [OF True ih])
  next
    case False
    then show ?thesis by (rule Cons-parallelI1)
  qed
qed

end

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

13 A small theory of prefix subtraction

The notion of *prefix-subtract* is need to make proofs more readable.

```
fun prefix-subtract :: 'a list ⇒ 'a list ⇒ 'a list (infix − 51)
where
  prefix-subtract []      xs      = []
| prefix-subtract (x#xs) []       = x#xs
| prefix-subtract (x#xs) (y#ys) = (if x = y then prefix-subtract xs ys else (x#xs))

lemma [simp]: (x @ y) − x = y
apply (induct x)
by (case-tac y, simp+)

lemma [simp]: x − x = []
by (induct x, auto)

lemma [simp]: x = xa @ y ⟹ x − xa = y
by (induct x, auto)

lemma [simp]: x − [] = x
by (induct x, auto)

lemma [simp]: (x − y = []) ⟹ (x ≤ y)
proof−
  have ∃ xa. x = xa @ (x − y) ∧ xa ≤ y
    apply (rule prefix-subtract.induct[of - x y], simp+)
    by (clarsimp, rule-tac x = y # xa in exI, simp+)
  thus (x − y = []) ⟹ (x ≤ y) by simp
qed

lemma diff-prefix:
  ⟦c ≤ a − b; b ≤ a⟧ ⟹ b @ c ≤ a
by (auto elim:prefixE)
```

lemma *diff-diff-appd*:
$[\![c < a - b;\ b < a]\!] \Longrightarrow (a - b) - c = a - (b\ @\ c)$
apply (*clarsimp simp*:*strict-prefix-def*)
by (*drule diff-prefix*, *auto elim*:*prefixE*)

lemma *app-eq-cases*[*rule-format*]:
$\forall\ x\ .\ x\ @\ y = m\ @\ n \longrightarrow (x \leq m \vee m \leq x)$
apply (*induct y*, *simp*)
apply (*clarify*, *drule-tac x* = *x* @ [*a*] **in** *spec*)
by (*clarsimp*, *auto simp*:*prefix-def*)

lemma *app-eq-dest*:
$x\ @\ y = m\ @\ n \Longrightarrow$
$(x \leq m \wedge (m - x)\ @\ n = y) \vee (m \leq x \wedge (x - m)\ @\ y = n)$
by (*frule-tac app-eq-cases*, *auto elim*:*prefixE*)

end

theory *Myhill-2*
imports *Myhill-1 List-Prefix Prefix-subtract*
begin

14 Direction *regular language* ⇒*finite partition*

14.1 The scheme

The following convenient notation $x \approx A\ y$ means: string x and y are equivalent with respect to language A.

definition
str-eq :: *string* ⇒ *lang* ⇒ *string* ⇒ *bool* (- ≈- -)
where
$x \approx A\ y \equiv (x,\ y) \in (\approx A)$

The main lemma (*rexp-imp-finite*) is proved by a structural induction over regular expressions. where base cases (cases for *NULL*, *EMPTY*, *CHAR*) are quite straightforward to proof. Real difficulty lies in inductive cases. By inductive hypothesis, languages defined by sub-expressions induce finite partitiions. Under such hypothsis, we need to prove that the language defined by the composite regular expression gives rise to finite partion. The basic idea is to attach a tag $tag(x)$ to every string x. The tagging fuction *tag* is carefully devised, which returns tags made of equivalent classes of the partitions induced by subexpressoins, and therefore has a finite range. Let *Lang* be the composite language, it is proved that:

If strings with the same tag are equivalent with respect to *Lang*, expressed as:

$$tag(x) = tag(y) \Longrightarrow x \approx Lang\ y$$

then the partition induced by *Lang* must be finite.

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

1. First, the tagging function *tag* induces an equivalent relation (=*tag*=) (defiintion of *f-eq-rel* and lemma *equiv-f-eq-rel*).
2. It is shown that: if the range of *tag* (denoted *range*(*tag*)) is finite, the partition given rise by (=*tag*=) is finite (lemma *finite-eq-f-rel*). Since tags are made from equivalent classes from component partitions, and the inductive hypothesis ensures the finiteness of these partitions, it is not difficult to prove the finiteness of *range*(*tag*).
3. It is proved that if equivalent relation *R1* is more refined than *R2* (expressed as $R1 \subseteq R2$), and the partition induced by *R1* is finite, then the partition induced by *R2* is finite as well (lemma *refined-partition-finite*).
4. The injectivity assumption $tag(x) = tag(y) \Longrightarrow x \approx Lang\ y$ implies that (=*tag*=) is more refined than (≈*Lang*).
5. Combining the points above, we have: the partition induced by language *Lang* is finite (lemma *tag-finite-imageD*).

definition
  f-eq-rel (=-=)
where
  =*f*= ≡ {(*x*, *y*) | *x y*. *f x* = *f y*}

lemma *equiv-f-eq-rel*:*equiv UNIV* (=*f*=)
  by (*auto simp*:*equiv-def f-eq-rel-def refl-on-def sym-def trans-def*)

lemma *finite-range-image*:
  assumes *finite* (*range f*)
  shows *finite* (*f* ‘ *A*)
  using *assms* **unfolding** *image-def*
  by (*rule-tac finite-subset*) (*auto*)

lemma *finite-eq-f-rel*:
  assumes *rng-fnt*: *finite* (*range tag*)
  shows *finite* (*UNIV* // =*tag*=)
proof −
  let *?f* = *op* ‘ *tag* **and** *?A* = (*UNIV* // =*tag*=)
  show *?thesis*
  proof (*rule-tac f* = *?f* **and** *A* = *?A* **in** *finite-imageD*)
    — The finiteness of *f*-image is a simple consequence of assumption *rng-fnt*:

show *finite* (*?f* ' *?A*)
proof −
have ∀ *X*. *?f X* ∈ (*Pow* (*range tag*)) **by** (*auto simp*:*image-def Pow-def*)
moreover from *rng-fnt* **have** *finite* (*Pow* (*range tag*)) **by** *simp*
ultimately have *finite* (*range ?f*)
by (*auto simp only*:*image-def intro*:*finite-subset*)
from *finite-range-image* [*OF this*] **show** *?thesis* **.**
qed
next
— The injectivity of *f*-image is a consequence of the definition of (=*tag*=):
show *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*
have *X* = *Y*
proof −
from *X-in Y-in tag-eq*
obtain *x y*
where *x-in*: *x* ∈ *X* **and** *y-in*: *y* ∈ *Y* **and** *eq-tg*: *tag x* = *tag y*
unfolding *quotient-def Image-def str-eq-rel-def*
str-eq-def image-def f-eq-rel-def
apply *simp* **by** *blast*
with *X-in Y-in* **show** *?thesis*
by (*auto simp*:*quotient-def str-eq-rel-def str-eq-def f-eq-rel-def*)
qed
} thus *?thesis* **unfolding** *inj-on-def* **by** *auto*
qed
qed
qed

lemma *finite-image-finite*:
⟦∀ *x* ∈ *A*. *f x* ∈ *B*; *finite B*⟧ ⟹ *finite* (*f* ' *A*)
by (*rule finite-subset* [*of* - *B*], *auto*)

lemma *refined-partition-finite*:
fixes *R1 R2 A*
assumes *fnt*: *finite* (*A* // *R1*)
and *refined*: *R1* ⊆ *R2*
and *eq1*: *equiv A R1* **and** *eq2*: *equiv A R2*
shows *finite* (*A* // *R2*)
proof −
let *?f* = λ *X*. {*R1* `` {*x*} | *x*. *x* ∈ *X*}
and *?A* = (*A* // *R2*) **and** *?B* = (*A* // *R1*)
show *?thesis*
proof(*rule-tac f* = *?f* **and** *A* = *?A* **in** *finite-imageD*)
show *finite* (*?f* ' *?A*)
proof(*rule finite-subset* [*of* - *Pow ?B*])

from *fnt* **show** *finite* (*Pow* (*A* // *R1*)) **by** *simp*
next
from *eq2*
show *?f* ' *A* // *R2* ⊆ *Pow ?B*
unfolding *image-def Pow-def quotient-def*
apply *auto*
by (*rule-tac x* = *xb* **in** *bexI*, *simp*,
unfold equiv-def sym-def refl-on-def, *blast*)
qed
next
show *inj-on ?f ?A*
proof –
{ fix *X Y*
assume *X-in*: *X* ∈ *?A* **and** *Y-in*: *Y* ∈ *?A*
and *eq-f*: *?f X* = *?f Y* (**is** *?L* = *?R*)
have *X* = *Y* **using** *X-in*
proof(*rule quotientE*)
fix *x*
assume *X* = *R2* " {*x*} **and** *x* ∈ *A* **with** *eq2*
have *x-in*: *x* ∈ *X*
unfolding *equiv-def quotient-def refl-on-def* **by** *auto*
with *eq-f* **have** *R1* " {*x*} ∈ *?R* **by** *auto*
then obtain *y* **where**
y-in: *y* ∈ *Y* **and** *eq-r*: *R1* " {*x*} = *R1* "{*y*} **by** *auto*
have (*x*, *y*) ∈ *R1*
proof –
from *x-in X-in y-in Y-in eq2*
have *x* ∈ *A* **and** *y* ∈ *A*
unfolding *equiv-def quotient-def refl-on-def* **by** *auto*
from *eq-equiv-class-iff* [*OF eq1 this*] **and** *eq-r*
show *?thesis* **by** *simp*
qed
with *refined* **have** *xy-r2*: (*x*, *y*) ∈ *R2* **by** *auto*
from *quotient-eqI* [*OF eq2 X-in Y-in x-in y-in this*]
show *?thesis* **.**
qed
} thus *?thesis* **by** (*auto simp*:*inj-on-def*)
qed
qed
qed

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

lemma *tag-finite-imageD*:
fixes *tag*
assumes *rng-fnt*: *finite* (*range tag*)
— Suppose the rang of tagging fucntion *tag* is finite.

```
  and same-tag-eqvt: ⋀ m n. tag m = tag (n::string) ⟹ m ≈Lang n
  — And strings with same tag are equivalent
  shows finite (UNIV // (≈Lang))
proof −
  let ?R1 = (=tag=)
  show ?thesis
  proof(rule-tac refined-partition-finite [of - ?R1])
    from finite-eq-f-rel [OF rng-fnt]
     show finite (UNIV // =tag=) .
   next
     from same-tag-eqvt
     show (=tag=) ⊆ (≈Lang)
       by (auto simp:f-eq-rel-def str-eq-def)
   next
     from equiv-f-eq-rel
     show equiv UNIV (=tag=) by blast
   next
     from equiv-lang-eq
     show equiv UNIV (≈Lang) by blast
  qed
qed
```

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

$$[\![finite\ (f\ `\ A);\ inj\text{-}on\ f\ A]\!] \Longrightarrow finite\ A$$

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

```
lemma
  fixes tag
  assumes rng-fnt: finite (range tag)
  — Suppose the rang of tagging fucntion tag is finite.
  and same-tag-eqvt: ⋀ m n. tag m = tag (n::string) ⟹ m ≈Lang n
  — And strings with same tag are equivalent
  shows finite (UNIV // (≈Lang))
  — Then the partition generated by (≈Lang) is finite.
proof −
  — The particular f and A used in finite-imageD are:
  let ?f = op ‘ tag and ?A = (UNIV // ≈Lang)
  show ?thesis
  proof (rule-tac f = ?f and A = ?A in finite-imageD)
    — The finiteness of f-image is a simple consequence of assumption rng-fnt:
    show finite (?f ‘ ?A)
    proof −
      have ∀ X. ?f X ∈ (Pow (range tag)) by (auto simp:image-def Pow-def)
      moreover from rng-fnt have finite (Pow (range tag)) by simp
      ultimately have finite (range ?f)
        by (auto simp only:image-def intro:finite-subset)
```

```
      from finite-range-image [OF this] show ?thesis .
    qed
  next
    — The injectivity of f is the consequence of assumption same-tag-eqvt:
    show 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
        have X = Y
        proof −
          from X-in Y-in tag-eq
         obtain x y where x-in: x ∈ X and y-in: y ∈ Y and eq-tg: tag x = tag y
            unfolding quotient-def Image-def str-eq-rel-def str-eq-def image-def
            apply simp by blast
          from same-tag-eqvt [OF eq-tg] have x ≈Lang y .
          with X-in Y-in x-in y-in
          show ?thesis by (auto simp:quotient-def str-eq-rel-def str-eq-def)
        qed
      } thus ?thesis unfolding inj-on-def by auto
    qed
  qed
qed
```

14.2 The proof

Each case is given in a separate section, as well as the final main lemma. Detailed explainations accompanied by illustrations are given for non-trivial cases.

For ever inductive case, there are two tasks, the easier one is to show the range finiteness of of the tagging function based on the finiteness of component partitions, the difficult one is to show that strings with the same tag are equivalent with respect to the composite language. Suppose the composite language be *Lang*, tagging function be *tag*, it amounts to show:

$$tag(x) = tag(y) \Longrightarrow x \approx Lang\ y$$

expanding the definition of $\approx Lang$, it amounts to show:

$$tag(x) = tag(y) \Longrightarrow (\forall\ z.\ x@z \in Lang \longleftrightarrow y@z \in Lang)$$

Because the assumed tag equlity $tag(x) = tag(y)$ is symmetric, it is suffcient to show just one direction:

$$\bigwedge x\ y\ z.\ [\![tag(x) = tag(y);\ x@z \in Lang]\!] \Longrightarrow y@z \in Lang$$

This is the pattern followed by every inductive case.

14.2.1 The base case for *NULL*

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

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

14.2.2 The base case for *EMPTY*

```
lemma quot-empty-subset:
  UNIV // (≈{[]}) ⊆ {{[]}, UNIV − {[]}}
proof
  fix x
  assume x ∈ UNIV // ≈{[]}
  then obtain y where h: x = {z. (y, z) ∈ ≈{[]}}
    unfolding quotient-def Image-def by blast
  show x ∈ {{[]}, UNIV − {[]}}
  proof (cases y = [])
    case True with h
    have x = {[]} by (auto simp: str-eq-rel-def)
    thus ?thesis by simp
  next
    case False with h
    have x = UNIV − {[]} by (auto simp: str-eq-rel-def)
    thus ?thesis by simp
  qed
qed

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

14.2.3 The base case for *CHAR*

```
lemma quot-char-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-rel-def)
    } moreover {
      assume y = [c] hence x = {[c]} using h
```

```
      by (auto dest!:spec[where x = []] simp:str-eq-rel-def)
    } moreover {
      assume y ≠ [] 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-rel-def)
    } ultimately show ?thesis by blast
  qed
qed

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

14.2.4 The inductive case for *ALT*

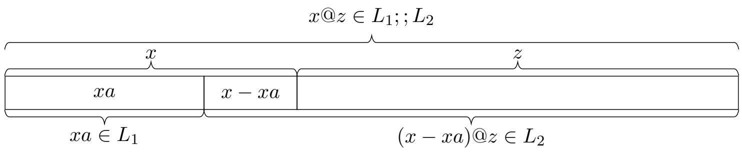
```
definition
  tag-str-ALT :: lang ⇒ lang ⇒ string ⇒ (lang × lang)
where
  tag-str-ALT L1 L2 ≡ (λx. (≈L1 `` {x}, ≈L2 `` {x}))

lemma quot-union-finiteI [intro]:
  fixes L1 L2::lang
  assumes finite1: finite (UNIV // ≈L1)
  and     finite2: finite (UNIV // ≈L2)
  shows finite (UNIV // ≈(L1 ∪ L2))
proof (rule-tac tag = tag-str-ALT L1 L2 in tag-finite-imageD)
  show ⋀x y. tag-str-ALT L1 L2 x = tag-str-ALT L1 L2 y ⟹ x ≈(L1 ∪ L2) y
    unfolding tag-str-ALT-def
    unfolding str-eq-def
    unfolding Image-def
    unfolding str-eq-rel-def
    by auto
next
  have *: finite ((UNIV // ≈L1) × (UNIV // ≈L2))
    using finite1 finite2 by auto
  show finite (range (tag-str-ALT L1 L2))
    unfolding tag-str-ALT-def
    apply(rule finite-subset[OF - *])
    unfolding quotient-def
    by auto
qed
```

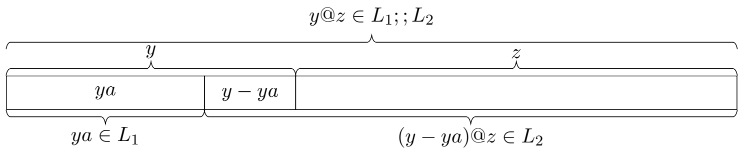
14.2.5 The inductive case for *SEQ*

For case *SEQ*, the language L is L_1 ;; L_2. Given x @ $z \in L_1$;; L_2, according to the defintion of L_1 ;; L_2, string x @ z can be splitted with the prefix in

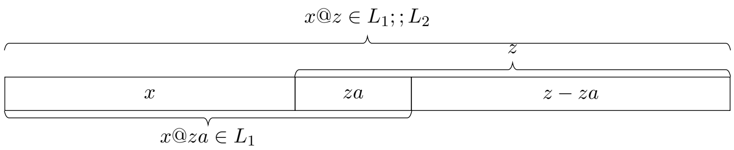
L_1 and suffix in L_2. The split point can either be in x (as shown in Fig. 1(a)), or in z (as shown in Fig. 1(c)). Whichever way it goes, the structure on x @ z cn be transfered faithfully onto y @ z (as shown in Fig. 1(b) and 1(d)) with the the help of the assumed tag equality. The following tag function *tag-str-SEQ* is such designed to facilitate such transfers and lemma *tag-str-SEQ-injI* formalizes the informal argument above. The details of structure transfer will be given their.



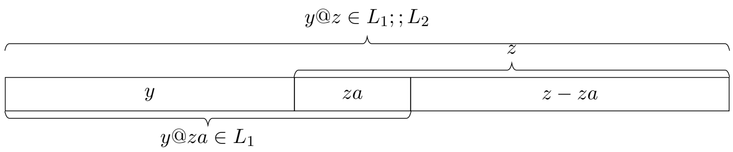
(a) First possible way to split $x@z$



(b) Transferred structure corresponding to the first way of splitting



(c) The second possible way to split $x@z$



(d) Transferred structure corresponding to the second way of splitting

Figure 1: The case for *SEQ*

definition
  tag-str-SEQ :: *lang* ⇒ *lang* ⇒ *string* ⇒ (*lang* × *lang set*)
where
  tag-str-SEQ L1 L2 ≡
    ($\lambda x.$ ($\approx$*L1* ‘‘ $\{x\}$, $\{(\approx$*L2* ‘‘ $\{x - xa\})$ | *xa*. *xa* $\leq x \wedge$ *xa* $\in$ *L1*$\}$))

The following is a techical lemma which helps to split the x @ $z \in L_1$;; L_2 mentioned above.

lemma *append-seq-elim*:
  assumes x @ $y \in L_1$;; L_2

shows $(\exists\ xa \leq x.\ xa \in L_1 \wedge (x - xa)\ @\ y \in L_2) \vee$
$(\exists\ ya \leq y.\ (x\ @\ ya) \in L_1 \wedge (y - ya) \in L_2)$
proof−
from *assms* **obtain** $s_1\ s_2$
where *eq-xys*: $x\ @\ y = s_1\ @\ s_2$
and *in-seq*: $s_1 \in L_1 \wedge s_2 \in L_2$
by (*auto simp*:*Seq-def*)
from *app-eq-dest* [*OF eq-xys*]
have
$(x \leq s_1 \wedge (s_1 - x)\ @\ s_2 = y) \vee (s_1 \leq x \wedge (x - s_1)\ @\ y = s_2)$
(**is** *?Split1* ∨ *?Split2*) **.**
moreover have *?Split1* $\Longrightarrow \exists\ ya \leq y.\ (x\ @\ ya) \in L_1 \wedge (y - ya) \in L_2$
using *in-seq* **by** (*rule-tac* $x = s_1 - x$ **in** *exI*, *auto elim*:*prefixE*)
moreover have *?Split2* $\Longrightarrow \exists\ xa \leq x.\ xa \in L_1 \wedge (x - xa)\ @\ y \in L_2$
using *in-seq* **by** (*rule-tac* $x = s_1$ **in** *exI*, *auto*)
ultimately show *?thesis* **by** *blast*
qed

lemma *tag-str-SEQ-injI*:
fixes *v w*
assumes *eq-tag*: *tag-str-SEQ* $L_1\ L_2\ v =$ *tag-str-SEQ* $L_1\ L_2\ w$
shows $v \approx (L_1\ ;;\ L_2)\ w$
proof−
— As explained before, a pattern for just one direction needs to be dealt with:
{ fix *x y z*
assume *xz-in-seq*: $x\ @\ z \in L_1\ ;;\ L_2$
and *tag-xy*: *tag-str-SEQ* $L_1\ L_2\ x =$ *tag-str-SEQ* $L_1\ L_2\ y$
have$y\ @\ z \in L_1\ ;;\ L_2$
proof−
— There are two ways to split $x@z$:
from *append-seq-elim* [*OF xz-in-seq*]
have $(\exists\ xa \leq x.\ xa \in L_1 \wedge (x - xa)\ @\ z \in L_2) \vee$
$(\exists\ za \leq z.\ (x\ @\ za) \in L_1 \wedge (z - za) \in L_2)$ **.**
— It can be shown that *?thesis* holds in either case:
moreover {
— The case for the first split:
fix *xa*
assume *h1*: $xa \leq x$ **and** *h2*: $xa \in L_1$ **and** *h3*: $(x - xa)\ @\ z \in L_2$
— The following subgoal implements the structure transfer:
obtain *ya*
where $ya \leq y$
and $ya \in L_1$
and $(y - ya)\ @\ z \in L_2$
proof −
By expanding the definition of

— *tag-str-SEQ* $L_1\ L_2\ x =$ *tag-str-SEQ* $L_1\ L_2\ y$

and extracting the second compoent, we get:

```
        have {≈L2 `` {x - xa} |xa. xa ≤ x ∧ xa ∈ L1} =
                {≈L2 `` {y - ya} |ya. ya ≤ y ∧ ya ∈ L1} (is ?Left = ?Right)
          using tag-xy unfolding tag-str-SEQ-def by simp
          — Since xa ≤ x and xa ∈ L1 hold, it is not difficult to show:
        moreover have ≈L2 `` {x - xa} ∈ ?Left using h1 h2 by auto
          — Through tag equality, equivalent class ≈L2 `` {x - xa}
            also belongs to the ?Right:
        ultimately have ≈L2 `` {x - xa} ∈ ?Right by simp
          — From this, the counterpart of xa in y is obtained:
        then obtain ya
          where eq-xya: ≈L2 `` {x - xa} = ≈L2 `` {y - ya}
          and pref-ya: ya ≤ y and ya-in: ya ∈ L1
          by simp blast
        — It can be proved that ya has the desired property:
        have (y - ya)@z ∈ L2
        proof -
          from eq-xya have (x - xa) ≈L2 (y - ya)
            unfolding Image-def str-eq-rel-def str-eq-def by auto
          with h3 show ?thesis unfolding str-eq-rel-def str-eq-def by simp
        qed
        — Now, ya has all properties to be a qualified candidate:
        with pref-ya ya-in
        show ?thesis using that by blast
      qed
        — From the properties of ya, y @ z ∈ L1 ;; L2 is derived easily.
      hence y @ z ∈ L1 ;; L2 by (erule-tac prefixE, auto simp:Seq-def)
    } moreover {
      — The other case is even more simpler:
      fix za
      assume h1: za ≤ z and h2: (x @ za) ∈ L1 and h3: z - za ∈ L2
      have y @ za ∈ L1
      proof-
        have ≈L1 `` {x} = ≈L1 `` {y}
          using tag-xy unfolding tag-str-SEQ-def by simp
        with h2 show ?thesis
          unfolding Image-def str-eq-rel-def str-eq-def by auto
      qed
      with h1 h3 have y @ z ∈ L1 ;; L2
        by (drule-tac A = L1 in seq-intro, auto elim:prefixE)
    }
    ultimately show ?thesis by blast
  qed
}
— ?thesis is proved by exploiting the symmetry of eq-tag:
from this [OF - eq-tag] and this [OF - eq-tag [THEN sym]]
  show ?thesis unfolding str-eq-def str-eq-rel-def by blast
qed

lemma quot-seq-finiteI [intro]:
```

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

14.2.6 The inductive case for *STAR*

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

1. Since x @ $z \in L_1*$ holds, a prefix xa of x can be found such that $xa \in L_1*$ and $(x - xa)@z \in L_1*$, as shown in Fig. 2(a). Such a prefix always exists, $xa = []$, for example, is one.

2. There could be many but fintie many of such xa, from which we can find the longest and name it *xa-max*, as shown in Fig. 2(b).

3. The next step is to split z into za and zb such that $(x - xa\text{-}max)$ @ $za \in L_1$ and $zb \in L_1*$ as shown in Fig. 2(e). Such a split always exists because:

   (a) Because $(x - x\text{-}max)$ @ $z \in L_1*$, it can always be splitted into prefix a and suffix b, such that $a \in L_1$ and $b \in L_1*$, as shown in Fig. 2(c).

   (b) But the prefix a CANNOT be shorter than $x - xa\text{-}max$ (as shown in Fig. 2(d)), becasue otherwise, $ma\text{-}max@a$ would be in the same kind as *xa-max* but with a larger size, conflicting with the fact that *xa-max* is the longest.

4. By the assumption that x and y have the same tag, the structure on x @ z can be transferred to y @ z as shown in Fig. 2(f). The detailed steps are:

   (a) A y-prefix ya corresponding to xa can be found, which satisfies conditions: $ya \in L_1*$ and $(y - ya)@za \in L_1$.

(b) Since we already know $zb \in L_1*$, we get $(y - ya)@za@zb \in L_1*$, and this is just $(y - ya)@z \in L_1*$.

(c) With fact $ya \in L_1*$, we finally get $y@z \in L_1*$.

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

definition
  tag-str-STAR :: *lang* ⇒ *string* ⇒ *lang set*
where
  tag-str-STAR L1 ≡ (λ*x*. {≈*L1* ‘‘ {*x* − *xa*} | *xa*. *xa* < *x* ∧ *xa* ∈ *L1*⋆})

A technical lemma.

lemma *finite-set-has-max*: ⟦*finite A*; *A* ≠ {}⟧ ⟹
      (∃ *max* ∈ *A*. ∀ *a* ∈ *A*. *f a* <= (*f max* :: *nat*))
proof (*induct rule*:*finite.induct*)
  case *emptyI* **thus** *?case* **by** *simp*
next
  case (*insertI A a*)
  show *?case*
  proof (*cases A* = {})
    case *True* **thus** *?thesis* **by** (*rule-tac x* = *a* **in** *bexI*, *auto*)
  next
    case *False*
    with *insertI.hyps* **and** *False*
    obtain *max*
      where *h1*: *max* ∈ *A*
      and *h2*: ∀ *a*∈*A*. *f a* ≤ *f max* **by** *blast*
    show *?thesis*
    proof (*cases f a* ≤ *f max*)
      assume *f a* ≤ *f max*
      with *h1 h2* **show** *?thesis* **by** (*rule-tac x* = *max* **in** *bexI*, *auto*)
    next
      assume ¬ (*f a* ≤ *f max*)
      thus *?thesis* **using** *h2* **by** (*rule-tac x* = *a* **in** *bexI*, *auto*)
    qed
  qed
qed

The following is a technical lemma.which helps to show the range finiteness of tag function.

lemma *finite-strict-prefix-set*: *finite* {*xa*. *xa* < (*x*::*string*)}
apply (*induct x rule*:*rev-induct*, *simp*)
apply (*subgoal-tac* {*xa*. *xa* < *xs* @ [*x*]} = {*xa*. *xa* < *xs*} ∪ {*xs*})
by (*auto simp*:*strict-prefix-def*)

lemma *tag-str-STAR-injI*:

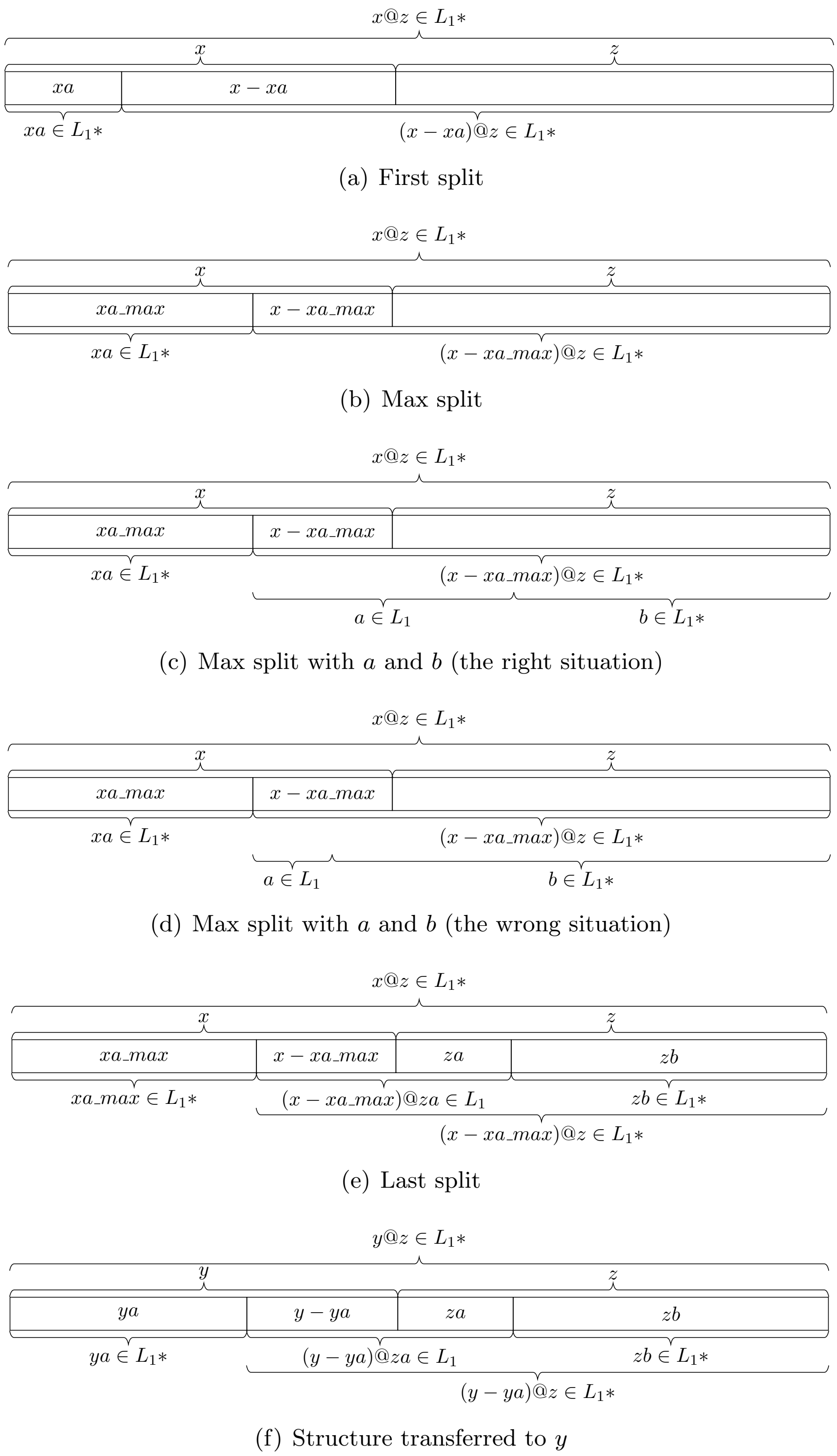


Figure 2: The case for $STAR$

fixes *v w*
assumes *eq-tag*: *tag-str-STAR* L_1 *v* = *tag-str-STAR* L_1 *w*
shows (*v*::*string*) $\approx$($L_1\star$) *w*
proof−
— As explained before, a pattern for just one direction needs to be dealt with:
{ **fix** *x y z*
assume *xz-in-star*: *x* @ *z* $\in L_1\star$
and *tag-xy*: *tag-str-STAR* L_1 *x* = *tag-str-STAR* L_1 *y*
have *y* @ *z* $\in L_1\star$
proof(*cases x* = [])
— The degenerated case when *x* is a null string is easy to prove:
case *True*
with *tag-xy* **have** *y* = []
by (*auto simp add*: *tag-str-STAR-def strict-prefix-def*)
thus *?thesis* **using** *xz-in-star True* **by** *simp*
next
— The nontrival case:
case *False*
— Since *x* @ *z* $\in L_1\star$, *x* can always be splitted by a prefix *xa* together with its suffix *x* − *xa*, such that both *xa* and (*x* − *xa*) @ *z* are in $L_1\star$, and there could be many such splittings.Therefore, the following set *?S* is nonempty, and finite as well:
let *?S* = {*xa*. *xa* < *x* ∧ *xa* $\in L_1\star$ ∧ (*x* − *xa*) @ *z* $\in L_1\star$}
have *finite ?S*
by (*rule-tac B* = {*xa*. *xa* < *x*} **in** *finite-subset*,
auto simp:*finite-strict-prefix-set*)
moreover have *?S* ≠ {} **using** *False xz-in-star*
by (*simp*, *rule-tac x* = [] **in** *exI*, *auto simp*:*strict-prefix-def*)
— Since *?S* is finite, we can always single out the longest and name it *xa-max*:
ultimately have ∃ *xa-max* ∈ *?S*. ∀ *xa* ∈ *?S*. *length xa* ≤ *length xa-max*
using *finite-set-has-max* **by** *blast*
then obtain *xa-max*
where *h1*: *xa-max* < *x*
and *h2*: *xa-max* $\in L_1\star$
and *h3*: (*x* − *xa-max*) @ *z* $\in L_1\star$
and *h4*:∀ *xa* < *x*. *xa* $\in L_1\star$ ∧ (*x* − *xa*) @ *z* $\in L_1\star$
⟶ *length xa* ≤ *length xa-max*
by *blast*
— By the equality of tags, the counterpart of *xa-max* among *y*-prefixes, named *ya*, can be found:
obtain *ya*
where *h5*: *ya* < *y* **and** *h6*: *ya* $\in L_1\star$
and *eq-xya*: (*x* − *xa-max*) $\approx L_1$ (*y* − *ya*)
proof−
from *tag-xy* **have** {$\approx L_1$ ‘‘ {*x* − *xa*} |*xa*. *xa* < *x* ∧ *xa* $\in L_1\star$} =
{$\approx L_1$ ‘‘ {*y* − *xa*} |*xa*. *xa* < *y* ∧ *xa* $\in L_1\star$} (**is** *?left* = *?right*)
by (*auto simp*:*tag-str-STAR-def*)
moreover have $\approx L_1$ ‘‘ {*x* − *xa-max*} ∈ *?left* **using** *h1 h2* **by** *auto*
ultimately have $\approx L_1$ ‘‘ {*x* − *xa-max*} ∈ *?right* **by** *simp*
thus *?thesis* **using** *that*

apply (*simp add*:*Image-def str-eq-rel-def str-eq-def*) **by** *blast*
qed
— The *?thesis*, $y @ z \in L_1\star$, is a simple consequence of the following proposition:
have $(y - ya) @ z \in L_1\star$
proof−
— The idea is to split the suffix z into za and zb, such that:
obtain *za zb* **where** *eq-zab*: $z = za @ zb$
and *l-za*: $(y - ya)@za \in L_1$ **and** *ls-zb*: $zb \in L_1\star$
proof −
— Since $xa\text{-}max < x$, x can be splitted into a and b such that:
from *h1* **have** $(x - xa\text{-}max) @ z \neq []$
by (*auto simp*:*strict-prefix-def elim*:*prefixE*)
from *star-decom* [*OF h3 this*]
obtain *a b* **where** *a-in*: $a \in L_1$
and *a-neq*: $a \neq []$ **and** *b-in*: $b \in L_1\star$
and *ab-max*: $(x - xa\text{-}max) @ z = a @ b$ **by** *blast*
— Now the candiates for za and zb are found:
let $?za = a - (x - xa\text{-}max)$ **and** $?zb = b$
have *pfx*: $(x - xa\text{-}max) \leq a$ (**is** *?P1*)
and *eq-z*: $z = ?za @ ?zb$ (**is** *?P2*)
proof −
— Since $(x - xa\text{-}max) @ z = a @ b$, string $(x - xa\text{-}max) @ z$ can be splitted in two ways:
have $((x - xa\text{-}max) \leq a \wedge (a - (x - xa\text{-}max)) @ b = z) \vee$
$(a < (x - xa\text{-}max) \wedge ((x - xa\text{-}max) - a) @ z = b)$
using *app-eq-dest*[*OF ab-max*] **by** (*auto simp*:*strict-prefix-def*)
moreover {
— However, the undsired way can be refuted by absurdity:
assume *np*: $a < (x - xa\text{-}max)$
and *b-eqs*: $((x - xa\text{-}max) - a) @ z = b$
have *False*
proof −
let $?xa\text{-}max' = xa\text{-}max @ a$
have $?xa\text{-}max' < x$
using *np h1* **by** (*clarsimp simp*:*strict-prefix-def diff-prefix*)
moreover have $?xa\text{-}max' \in L_1\star$
using *a-in h2* **by** (*simp add*:*star-intro3*)
moreover have $(x - ?xa\text{-}max') @ z \in L_1\star$
using *b-eqs b-in np h1* **by** (*simp add*:*diff-diff-appd*)
moreover have $\neg$ (*length* $?xa\text{-}max' \leq$ *length* *xa-max*)
using *a-neq* **by** *simp*
ultimately show *?thesis* **using** *h4* **by** *blast*
qed }
— Now it can be shown that the splitting goes the way we desired.
ultimately show *?P1* **and** *?P2* **by** *auto*
qed
hence $(x - xa\text{-}max)@?za \in L_1$ **using** *a-in* **by** (*auto elim*:*prefixE*)
— Now candidates *?za* and *?zb* have all the requred properteis.
with *eq-xya* **have** $(y - ya) @ ?za \in L_1$

by (*auto simp*:*str-eq-def str-eq-rel-def*)
with *eq-z* **and** *b-in*
show *?thesis* **using** *that* **by** *blast*
qed
— *?thesis* can easily be shown using properties of *za* and *zb*:
have $((y - ya) @ za) @ zb \in L_1\star$ **using** *l-za ls-zb* **by** *blast*
with *eq-zab* **show** *?thesis* **by** *simp*
qed
with *h5 h6* **show** *?thesis*
by (*drule-tac star-intro1*, *auto simp*:*strict-prefix-def elim*:*prefixE*)
qed
}
— By instantiating the reasoning pattern just derived for both directions:
from *this* [*OF* - *eq-tag*] **and** *this* [*OF* - *eq-tag* [*THEN sym*]]
— The thesis is proved as a trival consequence:
show *?thesis* **unfolding** *str-eq-def str-eq-rel-def* **by** *blast*
qed

lemma — The oringal version with less explicit details.
fixes *v w*
assumes *eq-tag*: *tag-str-STAR* L_1 *v* = *tag-str-STAR* L_1 *w*
shows $(v{::}string) \approx(L_1\star)\ w$
proof−
— According to the definition of ≈*Lang*, proving $v \approx(L_1\star)\ w$ amounts to showing: for any string *u*, if $v @ u \in (L_1\star)$ then $w @ u \in (L_1\star)$ and vice versa. The reasoning pattern for both directions are the same, as derived in the following:
{ fix *x y z*
assume *xz-in-star*: $x @ z \in L_1\star$
and *tag-xy*: *tag-str-STAR* L_1 *x* = *tag-str-STAR* L_1 *y*
have $y @ z \in L_1\star$
proof(*cases* *x* = [])
— The degenerated case when *x* is a null string is easy to prove:
case *True*
with *tag-xy* **have** *y* = []
by (*auto simp*:*tag-str-STAR-def strict-prefix-def*)
thus *?thesis* **using** *xz-in-star True* **by** *simp*
next
— The case when *x* is not null, and $x @ z$ is in $L_1\star$,
case *False*
obtain *x-max*
where *h1*: *x-max* < *x*
and *h2*: $x\text{-}max \in L_1\star$
and *h3*: $(x - x\text{-}max) @ z \in L_1\star$
and *h4*: $\forall\ xa < x.\ xa \in L_1\star \wedge (x - xa) @ z \in L_1\star \longrightarrow length\ xa \le length\ x\text{-}max$
proof−
let $?S = \{xa.\ xa < x \wedge xa \in L_1\star \wedge (x - xa) @ z \in L_1\star\}$
have *finite ?S*

```
    by (rule-tac B = {xa. xa < x} in finite-subset,
                      auto simp:finite-strict-prefix-set)
  moreover have ?S ≠ {} using False xz-in-star
    by (simp, rule-tac x = [] in exI, auto simp:strict-prefix-def)
  ultimately have ∃ max ∈ ?S. ∀ a ∈ ?S. length a ≤ length max
    using finite-set-has-max by blast
  thus ?thesis using that by blast
qed
obtain ya
  where h5: ya < y and h6: ya ∈ L₁⋆ and h7: (x − x-max) ≈L₁ (y − ya)
proof−
  from tag-xy have {≈L₁ `` {x − xa} |xa. xa < x ∧ xa ∈ L₁⋆} =
    {≈L₁ `` {y − xa} |xa. xa < y ∧ xa ∈ L₁⋆} (is ?left = ?right)
    by (auto simp:tag-str-STAR-def)
  moreover have ≈L₁ `` {x − x-max} ∈ ?left using h1 h2 by auto
  ultimately have ≈L₁ `` {x − x-max} ∈ ?right by simp
  with that show ?thesis apply
    (simp add:Image-def str-eq-rel-def str-eq-def) by blast
qed
have (y − ya) @ z ∈ L₁⋆
proof−
  from h3 h1 obtain a b where a-in: a ∈ L₁
    and a-neq: a ≠ [] and b-in: b ∈ L₁⋆
    and ab-max: (x − x-max) @ z = a @ b
    by (drule-tac star-decom, auto simp:strict-prefix-def elim:prefixE)
  have (x − x-max) ≤ a ∧ (a − (x − x-max)) @ b = z
  proof −
    have ((x − x-max) ≤ a ∧ (a − (x − x-max)) @ b = z) ∨
                    (a < (x − x-max) ∧ ((x − x-max) − a) @ z = b)
      using app-eq-dest[OF ab-max] by (auto simp:strict-prefix-def)
    moreover {
      assume np: a < (x − x-max) and b-eqs: ((x − x-max) − a) @ z = b
      have False
      proof −
        let ?x-max' = x-max @ a
        have ?x-max' < x
          using np h1 by (clarsimp simp:strict-prefix-def diff-prefix)
        moreover have ?x-max' ∈ L₁⋆
          using a-in h2 by (simp add:star-intro3)
        moreover have (x − ?x-max') @ z ∈ L₁⋆
          using b-eqs b-in np h1 by (simp add:diff-diff-appd)
        moreover have ¬ (length ?x-max' ≤ length x-max)
          using a-neq by simp
        ultimately show ?thesis using h4 by blast
      qed
    } ultimately show ?thesis by blast
  qed
  then obtain za where z-decom: z = za @ b
    and x-za: (x − x-max) @ za ∈ L₁
```

using *a-in* **by** (*auto elim*:*prefixE*)
from *x-za h7* **have** $(y - ya)$ @ $za \in L_1$
by (*auto simp*:*str-eq-def str-eq-rel-def*)
with *b-in* **have** $((y - ya)$ @ $za)$ @ $b \in L_1\star$ **by** *blast*
with *z-decom* **show** *?thesis* **by** *auto*
qed
with *h5 h6* **show** *?thesis*
by (*drule-tac star-intro1*, *auto simp*:*strict-prefix-def elim*:*prefixE*)
qed
}
— By instantiating the reasoning pattern just derived for both directions:
from *this* [*OF* - *eq-tag*] **and** *this* [*OF* - *eq-tag* [*THEN sym*]]
— The thesis is proved as a trival consequence:
show *?thesis* **unfolding** *str-eq-def str-eq-rel-def* **by** *blast*
qed

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

14.2.7 The conclusion

lemma *Myhill-Nerode2*:
fixes *r*::*rexp*
shows *finite* (*UNIV* // ≈(*L r*))
by (*induct r*) (*auto*)

end

theory *Myhill*
imports *Myhill-2*
begin

15 Preliminaries

15.1 Finite automata and Myhill-Nerode theorem

A *determinisitc finite automata (DFA)* M is a 5-tuple $(Q, \Sigma, \delta, s, F)$, where:

1. Q is a finite set of *states*, also denoted Q_M.
2. Σ is a finite set of *alphabets*, also denoted Σ_M.
3. δ is a *transition function* of type $Q \times \Sigma \Rightarrow Q$ (a total function), also denoted δ_M.
4. $s \in Q$ is a state called *initial state*, also denoted s_M.
5. $F \subseteq Q$ is a set of states named *accepting states*, also denoted F_M.

Therefore, we have $M = (Q_M, \Sigma_M, \delta_M, s_M, F_M)$. Every DFA M can be interpreted as a function assigning states to strings, denoted $\hat{\delta}_M$, the definition of which is as the following:

$$\begin{aligned} \hat{\delta}_M([]) &\equiv s_M \\ \hat{\delta}_M(xa) &\equiv \delta_M(\hat{\delta}_M(x), a) \end{aligned} \tag{1}$$

A string x is said to be *accepted* (or *recognized*) by a DFA M if $\hat{\delta}_M(x) \in F_M$. The language recoginzed by DFA M, denoted $L(M)$, is defined as:

$$L(M) \equiv \{x \mid \hat{\delta}_M(x) \in F_M\} \tag{2}$$

The standard way of specifying a laugage $\mathcal{L}$ as *regular* is by stipulating that: $\mathcal{L} = L(M)$ for some DFA M.

For any DFA M, the DFA obtained by changing initial state to another $p \in Q_M$ is denoted M_p, which is defined as:

$$M_p \;\equiv\; (Q_M, \Sigma_M, \delta_M, p, F_M) \tag{3}$$

Two states $p, q \in Q_M$ are said to be *equivalent*, denoted $p \approx_M q$, iff.

$$L(M_p) = L(M_q) \tag{4}$$

It is obvious that $\approx_M$ is an equivalent relation over Q_M. and the partition induced by $\approx_M$ has $|Q_M|$ equivalent classes. By overloading $\approx_M$, an equivalent relation over strings can be defined:

$$x \approx_M y \;\;\equiv\;\; \hat{\delta}_M(x) \approx_M \hat{\delta}_M(y) \tag{5}$$

It can be proved that the the partition induced by $\approx_M$ also has $|Q_M|$ equivalent classes. It is also easy to show that: if $x \approx_M y$, then $x \approx_{L(M)} y$, and this means $\approx_M$ is a more refined equivalent relation than $\approx_{L(M)}$. Since partition induced by $\approx_M$ is finite, the one induced by $\approx_{L(M)}$ must also be finite, and this is one of the two directions of Myhill-Nerode theorem:

Lemma 1 (Myhill-Nerode theorem, Direction two)**.** *If a language $\mathcal{L}$ is regular (i.e. $\mathcal{L} = L(M)$ for some DFA M), then the partition induced by $\approx_{\mathcal{L}}$ is finite.*

The other direction is:

Lemma 2 (Myhill-Nerode theorem, Direction one)**.** *If the partition induced by $\approx_{\mathcal{L}}$ is finite, then $\mathcal{L}$ is regular (i.e. $\mathcal{L} = L(M)$ for some DFA M).*

The M we are seeking when prove lemma **??** can be constructed out of $\approx_{\mathcal{L}}$, denoted $M_{\mathcal{L}}$ and defined as the following:

$$Q_{M_{\mathcal{L}}} \equiv \{[\![x]\!]_{\approx_{\mathcal{L}}} \mid x \in \Sigma^*\} \tag{6a}$$
$$\Sigma_{M_{\mathcal{L}}} \equiv \Sigma_M \tag{6b}$$
$$\delta_{M_{\mathcal{L}}} \equiv (\lambda([\![x]\!]_{\approx_{\mathcal{L}}}, a).[\![xa]\!]_{\approx_{\mathcal{L}}}) \tag{6c}$$
$$s_{M_{\mathcal{L}}} \equiv [\![[]]\!]_{\approx_{\mathcal{L}}} \tag{6d}$$
$$F_{M_{\mathcal{L}}} \equiv \{[\![x]\!]_{\approx_{\mathcal{L}}} \mid x \in \mathcal{L}\} \tag{6e}$$

It can be proved that $Q_{M_{\mathcal{L}}}$ is indeed finite and $\mathcal{L} = L(M_{\mathcal{L}})$, so lemma 2 holds. It can also be proved that $M_{\mathcal{L}}$ is the minimal DFA (therefore unique) which recoginzes $\mathcal{L}$.

15.2 The objective and the underlying intuition

It is now obvious from section 15.1 that Myhill-Nerode theorem can be established easily when *reglar languages* are defined as ones recognized by finite automata. Under the context where the use of finite automata is forbiden, the situation is quite different. The theorem now has to be expressed as:

Theorem 1 (Myhill-Nerode theorem, Regular expression version)**.** *A language $\mathcal{L}$ is regular (i.e. $\mathcal{L} = L(e)$ for some* regular expression *e) iff. the partition induced by $\approx_{\mathcal{L}}$ is finite.*

The proof of this version consists of two directions (if the use of automata are not allowed):

Direction one: generating a regular expression e out of the finite partition induced by $\approx_{\mathcal{L}}$, such that $\mathcal{L} = L(e)$.

Direction two: showing the finiteness of the partition induced by $\approx_{\mathcal{L}}$, under the assmption that $\mathcal{L}$ is recognized by some regular expression e (i.e. $\mathcal{L} = L(e)$).

The development of these two directions consititutes the body of this paper.

16 Direction *regular language ⇒finite partition*

Although not used explicitly, the notion of finite autotmata and its relationship with language partition, as outlined in section 15.1, still servers as important intuitive guides in the development of this paper. For example, *Direction one* follows the *Brzozowski algebraic method* used to convert finite autotmata to regular expressions, under the intuition that every partition member $[\![x]\!]_{\approx_{\mathcal{L}}}$ is a state in the DFA $M_{\mathcal{L}}$ constructed to prove lemma 2 of section 15.1.

The basic idea of Brzozowski method is to extract an equational system out of the transition relationship of the automaton in question. In the equational system, every automaton state is represented by an unknown, the solution of which is expected to be a regular expresion characterizing the state in a certain sense. There are two choices of how a automaton state can be characterized. The first is to characterize by the set of strings leading from the state in question into accepting states. The other choice is to characterize by the set of strings leading from initial state into the state in question. For the second choice, the language recognized the automaton can be characterized by the solution of initial state, while for the second choice, the language recoginzed by the automaton can be characterized by combining solutions of all accepting states by +. Because of the automaton used as our intuitive guide, the $M_{\mathcal{L}}$, the states of which are sets of strings leading from initial state, the second choice is used in this paper.

Supposing the automaton in Fig 3 is the $M_{\mathcal{L}}$ for some language $\mathcal{L}$, and suppose $\Sigma = \{a, b, c, d, e\}$. Under the second choice, the equational system extracted is:

$$X_0 = X_1 \cdot c + X_2 \cdot d + \lambda \tag{7a}$$

$$X_1 = X_0 \cdot a + X_1 \cdot b + X_2 \cdot d \tag{7b}$$

$$X_2 = X_0 \cdot b + X_1 \cdot d + X_2 \cdot a \tag{7c}$$

$$X_3 = \begin{array}{l} X_0 \cdot (c + d + e) + X_1 \cdot (a + e) + X_2 \cdot (b + e) + \\ X_3 \cdot (a + b + c + d + e) \end{array} \tag{7d}$$

Every ·-item on the right side of equations describes some state transtions, except the λ in (7a), which represents empty string []. The reason is that: every state is characterized by the set of incoming strings leading from initial state. For non-initial state, every such string can be splitted into a prefix leading into a preceding state and a single character suffix transiting into from the preceding state. The exception happens at initial state, where the empty string is a incoming string which can not be splitted. The λ in (7a) is introduce to repsent this indivisible string. There is one and only one λ in every equational system such obtained, becasue [] can only be contaied in one equivalent class (the intial state in $M_{\mathcal{L}}$) and equivalent classes are disjoint.

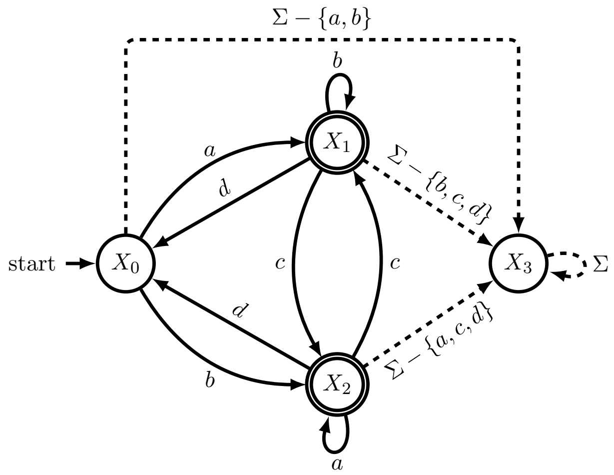


Figure 3: An example automaton

Suppose all unknowns (X_0, X_1, X_2, X_3) are solvable, the regular expression charactering laugnage $\mathcal{L}$ is $X_1 + X_2$. This paper gives a procedure by which arbitrarily picked unknown can be solved. The basic idea to solve X_i is by eliminating all variables other than X_i from the equational system. If X_0 is the one picked to be solved, variables X_1, X_2, X_3 have to be removed one by one. The order to remove does not matter as long as the remaing equations are kept valid. Suppose X_1 is the first one to remove, the action is to replace all occurences of X_1 in remaining equations by the right hand side of its characterizing equation, i.e. the $X_0 \cdot a + X_1 \cdot b + X_2 \cdot d$ in (7b). However, because of the recursive occurence of X_1, this replacement does not really removed X_1. Arden's lemma is invoked to transform recursive equations like (7b) into non-recursive ones. For example, the recursive equation (7b) is transformed into the follwing non-recursive one:

$$X_1 = (X_0 \cdot a + X_2 \cdot d) \cdot b^* = X_0 \cdot (a \cdot b^*) + X_2 \cdot (d \cdot b^*) \tag{8}$$

which, by Arden's lemma, still characterizes X_1 correctly. By subsituting $(X_0 \cdot a + X_2 \cdot d) \cdot b^*$ for all X_1 and removing (7b), we get:

$$X_0 = \begin{array}{l} (X_0 \cdot (a \cdot b^*) + X_2 \cdot (d \cdot b^*)) \cdot c + X_2 \cdot d + \lambda = \\ X_0 \cdot (a \cdot b^* \cdot c) + X_2 \cdot (d \cdot b^* \cdot c) + X_2 \cdot d + \lambda = \\ X_0 \cdot (a \cdot b^* \cdot c) + X_2 \cdot (d \cdot b^* \cdot c + d) + \lambda \end{array} \tag{9a}$$

$$X_2 = \begin{array}{l} X_0 \cdot b + (X_0 \cdot (a \cdot b^*) + X_2 \cdot (d \cdot b^*)) \cdot d + X_2 \cdot a = \\ X_0 \cdot b + X_0 \cdot (a \cdot b^* \cdot d) + X_2 \cdot (d \cdot b^* \cdot d) + X_2 \cdot a = \\ X_0 \cdot (b + a \cdot b^* \cdot d) + X_2 \cdot (d \cdot b^* \cdot d + a) \end{array} \tag{9b}$$

$$X_3 = \begin{array}{l} X_0 \cdot (c + d + e) + ((X_0 \cdot a + X_2 \cdot d) \cdot b^*) \cdot (a + e) \\ + X_2 \cdot (b + e) + X_3 \cdot (a + b + c + d + e) \end{array} \tag{9c}$$

Suppose X_3 is the one to remove next, since X_3 dose not appear in either X_0 or X_2, the removal of equation (9c) changes nothing in the rest equations. Therefore, we get:

$$
\begin{aligned}
X_0 &= X_0 \cdot (a \cdot b^* \cdot c) + X_2 \cdot (d \cdot b^* \cdot c + d) + \lambda && \text{(10a)} \\
X_2 &= X_0 \cdot (b + a \cdot b^* \cdot d) + X_2 \cdot (d \cdot b^* \cdot d + a) && \text{(10b)}
\end{aligned}
$$

Actually, since absorbing state like X_3 contributes nothing to the language $\mathcal{L}$, it could have been removed at the very beginning of this precedure without affecting the final result. Now, the last unknown to remove is X_2 and the Arden's transformaton of (10b) is:

$$
X_2 = (X_0 \cdot (b + a \cdot b^* \cdot d)) \cdot (d \cdot b^* \cdot d + a)^* = X_0 \cdot ((b + a \cdot b^* \cdot d) \cdot (d \cdot b^* \cdot d + a)^*) \quad (11)
$$

By substituting the right hand side of (11) into (10a), we get:

$$
\begin{aligned}
X_0 = \; & X_0 \cdot (a \cdot b^* \cdot c) + \\
& X_0 \cdot ((b + a \cdot b^* \cdot d) \cdot (d \cdot b^* \cdot d + a)^*) \cdot (d \cdot b^* \cdot c + d) + \lambda \\
= \; & X_0 \cdot ((a \cdot b^* \cdot c) + \\
& \quad ((b + a \cdot b^* \cdot d) \cdot (d \cdot b^* \cdot d + a)^*) \cdot (d \cdot b^* \cdot c + d)) + \lambda
\end{aligned} \quad (12)
$$

By applying Arden's transformation to this, we get the solution of X_0 as:

$$
X_0 = ((a \cdot b^* \cdot c) + ((b + a \cdot b^* \cdot d) \cdot (d \cdot b^* \cdot d + a)^*) \cdot (d \cdot b^* \cdot c + d))^* \quad (13)
$$

Using the same method, solutions for X_1 and X_2 can be obtained as well and the regular expressoin for $\mathcal{L}$ is just $X_1 + X_2$. The formalization of this procedure consitiutes the first direction of the *regular expression* verion of Myhill-Nerode theorem. Detailed explaination are given in **paper.pdf** and more details and comments can be found in the formal scripts.

17 Direction *finite partition* $\Rightarrow$ *regular language*

It is well known in the theory of regular languages that the existence of finite language partition amounts to the existence of a minimal automaton, i.e. the $M_{\mathcal{L}}$ constructed in section 15, which recoginzes the same language $\mathcal{L}$. The standard way to prove the existence of finite language partition is to construct a automaton out of the regular expression which recoginzes the same language, from which the existence of finite language partition follows immediately. As discussed in the introducton of **paper.pdf** as well as in [5], the problem for this approach happens when automata of sub regular expressions are combined to form the automaton of the mother regular expression, no matter what kind of representation is used, the formalization is cubersome, as said by Nipkow in [5]: '*a more abstract mathod is clearly desirable*'.

In this section, an *intrinsically abstract* method is given, which completely avoid the mentioning of automata, let along any particular representations.

The main proof structure is a structural induction on regular expressions, where base cases (cases for *NULL*, *EMPTY*, *CHAR*) are quite straightforward to proof. Real difficulty lies in inductive cases. By inductive hypothesis, languages defined by sub-expressions induce finite partitiions. Under such hypothsis, we need to prove that the language defined by the composite regular expression gives rise to finite partion. The basic idea is to attach a tag $tag(x)$ to every string x. The tagging fuction tag is carefully devised, which returns tags made of equivalent classes of the partitions induced by subexpressoins, and therefore has a finite range. Let *Lang* be the composite language, it is proved that:

> If strings with the same tag are equivalent with respect to *Lang*, expressed as:
>
> $$tag(x) = tag(y) \Longrightarrow x \approx Lang\ y$$
>
> then the partition induced by *Lang* must be finite.

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

1. First, the tagging function tag induces an equivalent relation $(=tag=)$ (defiintion of *f-eq-rel* and lemma *equiv-f-eq-rel*).
2. It is shown that: if the range of tag (denoted $range(tag)$) is finite, the partition given rise by $(=tag=)$ is finite (lemma *finite-eq-f-rel*). Since tags are made from equivalent classes from component partitions, and the inductive hypothesis ensures the finiteness of these partitions, it is not difficult to prove the finiteness of $range(tag)$.
3. It is proved that if equivalent relation $R1$ is more refined than $R2$ (expressed as $R1 \subseteq R2$), and the partition induced by $R1$ is finite, then the partition induced by $R2$ is finite as well (lemma *refined-partition-finite*).
4. The injectivity assumption $tag(x) = tag(y) \Longrightarrow x \approx Lang\ y$ implies that $(=tag=)$ is more refined than $(\approx Lang)$.
5. Combining the points above, we have: the partition induced by language *Lang* is finite (lemma *tag-finite-imageD*).

We could have followed another approach given in appendix II of Brzozowski's paper [?], where the set of derivatives of any regular expression can be proved to be finite. Since it is easy to show that strings with same derivative are equivalent with respect to the language, then the second direction follows. We believe that our apporoach is easy to formalize, with no need to do complicated derivation calculations and countings as in [???].

end