tphols-2011

By xingyuan

February 7, 2011

Contents

1 List prefixes and postfixes

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

1.1 Prefix order on lists

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

definition
  prefix-def: xs ≤ ys ⟷ (∃ zs. ys = xs @ zs)

definition
  strict-prefix-def: xs < ys ⟷ xs ≤ ys ∧ xs ≠ (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*

1.2 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* ≤ *y* # *ys*) = (*x* = *y* ∧ *xs* ≤ *ys*)
by (*auto simp add*: *prefix-def*)

lemma *less-eq-list-code* [*code*]:
([]::'*a*::{*equal*, *ord*} *list*) ≤ *xs* ⟷ *True*
(*x*::'*a*::{*equal*, *ord*}) # *xs* ≤ [] ⟷ *False*
(*x*::'*a*::{*equal*, *ord*}) # *xs* ≤ *y* # *ys* ⟷ *x* = *y* ∧ *xs* ≤ *ys*
by *simp-all*

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

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

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

lemma *append-prefixD*: *xs* @ *ys* ≤ *zs* ⟹ *xs* ≤ *zs*
by (*auto simp add*: *prefix-def*)

theorem *prefix-Cons*: (*xs* ≤ *y* # *ys*) = (*xs* = [] ∨ (∃ *zs*. *xs* = *y* # *zs* ∧ *zs* ≤ *ys*))

by (*cases xs*) (*auto simp add*: *prefix-def*)

theorem *prefix-append*:
$(xs \le ys\ @\ zs) = (xs \le ys \lor (\exists\, us.\ xs = ys\ @\ us \land us \le zs))$
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*:
$xs \le ys ==> length\ xs < length\ ys ==> xs\ @\ [ys\ !\ length\ xs] \le ys$
unfolding *prefix-def*
by (*metis Cons-eq-appendI append-eq-appendI append-eq-conv-conj eq-Nil-appendI nth-drop'*)

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

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

lemma *set-mono-prefix*: $xs \le ys \Longrightarrow set\ xs \subseteq set\ ys$
by (*auto simp add*: *prefix-def*)

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

lemma *map-prefixI*: $xs \le ys \Longrightarrow map\ f\ xs \le map\ f\ ys$
by (*auto simp*: *prefix-def*)

lemma *prefix-length-less*: $xs < ys \Longrightarrow length\ xs < length\ ys$
by (*auto simp*: *strict-prefix-def prefix-def*)

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

lemma *take-strict-prefix*: $xs < ys \Longrightarrow take\ n\ xs < ys$
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*: $\neg\ ps \le ls$

```
  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
```

1.3 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*: $xs \parallel ys \Longrightarrow x = xs\ @\ xs' \Longrightarrow y = ys\ @\ ys' \Longrightarrow x \parallel y$
  by (*simp add*: *parallel-append*)

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

1.4 Postfix order on lists

definition
  postfix :: *'a list => 'a list => bool* ((-/ >>= -) [*51*, *50*] *50*) **where**
  (*xs* >>= *ys*) = (∃ *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
```

2 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*]: $x = xa\ @\ y \Longrightarrow x - xa = y$
by (*induct x*, *auto*)

lemma [*simp*]: $x - [] = x$
by (*induct x*, *auto*)

lemma [*simp*]: $(x - y = []) \Longrightarrow (x \le y)$
proof−
  have $\exists\, xa.\ x = xa\ @\ (x - y) \wedge xa \le y$
    apply (*rule prefix-subtract.induct*[*of - x y*], *simp*+)
    by (*clarsimp*, *rule-tac* $x = y\ \#\ xa$ **in** *exI*, *simp*+)
  thus $(x - y = []) \Longrightarrow (x \le y)$ **by** *simp*
qed

lemma *diff-prefix*:
  $[\![c \le a - b;\ b \le a]\!] \Longrightarrow b\ @\ c \le a$
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*]:
  $\forall\ x\ .\ x\ @\ y = m\ @\ n \longrightarrow (x \le m \vee m \le x)$
apply (*induct y*, *simp*)
apply (*clarify*, *drule-tac* $x = x\ @\ [a]$ **in** *spec*)
by (*clarsimp*, *auto simp*:*prefix-def*)

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

end

theory *Prelude*
imports *Main*
begin

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

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

3 Preliminary definitions

types *lang* = *string set*

Sequential composition of two languages

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
apply(induct rule: star-induct)
apply(simp)
apply(blast)
done

lemma
  shows seq-Union-left:  B ;; (⋃n. A ↑ n) = (⋃n. B ;; (A ↑ n))
  and   seq-Union-right: (⋃n. A ↑ n) ;; B = (⋃n. (A ↑ n) ;; B)
unfolding Seq-def by auto

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

```
lemma seq-star-comm:
  shows A ;; A⋆ = A⋆ ;; A
unfolding Star-def
unfolding seq-Union-left
unfolding seq-pow-comm
unfolding seq-Union-right
by simp
```

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

```
lemma pow-length:
  assumes a: [] ∉ A
  and     b: s ∈ A ↑ Suc n
  shows n < length s
using b
proof (induct n arbitrary: s)
  case 0
  have s ∈ A ↑ Suc 0 by fact
  with a have s ≠ [] by auto
  then show 0 < length s by auto
next
  case (Suc n)
  have ih: ⋀s. s ∈ A ↑ Suc n ⟹ n < length s by fact
  have s ∈ A ↑ Suc (Suc n) by fact
  then obtain s1 s2 where eq: s = s1 @ s2 and ∗: s1 ∈ A and ∗∗: s2 ∈ A ↑
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: [] ∉ A
  and     b: s ∈ B ;; (A ↑ Suc n)
  shows n < length s
proof −
  from b obtain s1 s2 where eq: s = s1 @ s2 and ∗: s2 ∈ A ↑ 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 slightly modified version of Arden's lemma

A helper lemma for Arden

```
lemma ardens-helper:
```

```
  assumes eq: X = X ;; A ∪ B
  shows X = X ;; (A ↑ Suc n) ∪ (⋃m∈{0..n}. B ;; (A ↑ m))
proof (induct n)
  case 0
  show X = X ;; (A ↑ Suc 0) ∪ (⋃(m::nat)∈{0..0}. B ;; (A ↑ m))
    using eq by simp
next
  case (Suc n)
  have ih: X = X ;; (A ↑ Suc n) ∪ (⋃m∈{0..n}. B ;; (A ↑ m)) by fact
  also have ... = (X ;; A ∪ B) ;; (A ↑ Suc n) ∪ (⋃m∈{0..n}. B ;; (A ↑ m))
using eq by simp
  also have ... = X ;; (A ↑ Suc (Suc n)) ∪ (B ;; (A ↑ Suc n)) ∪ (⋃m∈{0..n}.
B ;; (A ↑ m))
    by (simp add: seq-union-distrib-right seq-assoc)
  also have ... = X ;; (A ↑ Suc (Suc n)) ∪ (⋃m∈{0..Suc n}. B ;; (A ↑ m))
    by (auto simp add: le-Suc-eq)
  finally show X = X ;; (A ↑ Suc (Suc n)) ∪ (⋃m∈{0..Suc n}. B ;; (A ↑ m)) .
qed

theorem ardens-revised:
  assumes nemp: [] ∉ A
  shows X = X ;; A ∪ B ⟷ X = B ;; A⋆
proof
  assume eq: X = B ;; A⋆
  have A⋆ = {[]} ∪ A⋆ ;; A
    unfolding seq-star-comm[symmetric]
    by (rule star-cases)
  then have B ;; A⋆ = B ;; ({[]} ∪ A⋆ ;; A)
    by (rule seq-add-left)
  also have ... = B ∪ B ;; (A⋆ ;; A)
    unfolding seq-union-distrib-left by simp
  also have ... = B ∪ (B ;; A⋆) ;; A
    by (simp only: seq-assoc)
  finally show X = X ;; A ∪ B
    using eq by blast
next
  assume eq: X = X ;; A ∪ B
  { fix n::nat
    have B ;; (A ↑ n) ⊆ X using ardens-helper[OF eq, of n] by auto }
  then have B ;; A⋆ ⊆ X
    unfolding Seq-def Star-def UNION-def
    by auto
  moreover
  { fix s::string
    obtain k where k = length s by auto
    then have not-in: s ∉ X ;; (A ↑ 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 ardens-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 following *L* is an overloaded operator, where $L(x)$ evaluates to the language represented by the syntactic object x.

```
consts L:: 'a ⇒ lang
```

The *L* (*rexp*) for regular expressions.

```
overloading L-rexp ≡ L:: rexp ⇒ lang
begin
fun
  L-rexp :: rexp ⇒ string set
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
```

6 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-graph f z S x*

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

lemma *folds-alt-simp* [*simp*]:
assumes *a*: *finite rs*
shows *L* (*folds ALT NULL rs*) = $\bigcup$ (*L* ‘ *rs*)
apply(*rule set-eq-intro*)
apply(*simp add*: *folds-def*)
apply(*rule someI2-ex*)
apply(*rule-tac finite-imp-fold-graph*[*OF a*])
apply(*erule fold-graph.induct*)
apply(*auto*)
done

Just a technical lemma for collections and pairs

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

$\approx A$ is an equivalence class defined by language A.

definition
str-eq-rel :: *lang* $\Rightarrow$ (*string* $\times$ *string*) *set* ($\approx$- [*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* $\Rightarrow$ *lang set*
where
finals A $\equiv \{\approx A$ ‘‘ $\{x\}\ |\ x\ .\ x \in A\}$

The following lemma establishes the relationshipt between *finals A* and *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

7 Direction *finite partition* ⇒ *regular language*

The relationship between equivalent classes can be described by an equational system. For example, in equational system (1), X_0, X_1 are equivalent classes. The first equation says every string in X_0 is obtained either by appending one b to a string in X_0 or by appending one a to a string in X_1 or just be an empty string (represented by the regular expression λ). Similary, the second equation tells how the strings inside X_1 are composed.

$$\begin{aligned} X_0 &= X_0 b + X_1 a + \lambda \\ X_1 &= X_0 a + X_1 b \end{aligned} \tag{1}$$

The summands on the right hand side is represented by the following data type *rhs-item*, mnemonic for 'right hand side item'. Generally, there are two kinds of right hand side items, one kind corresponds to pure regular expressions, like the λ in (1), the other kind corresponds to transitions from one one equivalent class to another, like the $X_0 b, X_1 a$ etc.

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

In this formalization, pure regular expressions like λ is repsented by $Lam(EMPTY)$, while transitions like $X_0 a$ is represented by $Trn\ X_0\ (CHAR\ a)$.

The functions *the-r* and *the-Trn* are used to extract subcomponents from right hand side items.

fun
  the-r :: *rhs-item* ⇒ *rexp*
where
  the-r (*Lam r*) = *r*

fun
  the-Trn:: *rhs-item* ⇒ (*lang* × *rexp*)
where
  the-Trn (*Trn Y r*) = (*Y*, *r*)

Every right-hand side item *itm* defines a language given by $L(itm)$, defined as:

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

The right hand side of every equation is represented by a set of items. The string set defined by such a set *itms* is given by $L(itms)$, defined as:

overloading *L-rhs* ≡ *L*:: *rhs-item set* ⇒ *lang*
begin
  fun *L-rhs*:: *rhs-item set* ⇒ *lang*
  where
    $L\text{-}rhs\ rhs = \bigcup\ (L\ `\ rhs)$
end

Given a set of equivalence classes *CS* and one equivalence class *X* among *CS*, the term *init-rhs CS X* is used to extract the right hand side of the equation describing the formation of *X*. The definition of *init-rhs* is:

definition
  $transition :: lang \Rightarrow char \Rightarrow lang \Rightarrow bool$ (- $\models$-$\Rightarrow$- [*100*,*100*,*100*] *100*)
where
  $Y \models c \Rightarrow X \equiv Y\ ;;\ \{[c]\} \subseteq X$

definition
  init-rhs CS X ≡
    if $([] \in X)$ *then*
      $\{Lam\ EMPTY\} \cup \{Trn\ Y\ (CHAR\ c) \mid Y\ c.\ Y \in CS \wedge Y \models c \Rightarrow X\}$
    else
      $\{Trn\ Y\ (CHAR\ c) \mid Y\ c.\ Y \in CS \wedge Y \models c \Rightarrow X\}$

In the definition of *init-rhs*, the term $\{Trn\ Y\ (CHAR\ c) \mid Y\ c.\ Y \in CS \wedge Y\ ;;\ \{[c]\} \subseteq X\}$ appearing on both branches describes the formation of strings in *X* out of transitions, while the term $\{Lam(EMPTY)\}$ describes the empty string which is intrinsically contained in *X* rather than by transition. This $\{Lam(EMPTY)\}$ corresponds to the λ in (1).

With the help of *init-rhs*, the equitional system descrbing the formation of every equivalent class inside *CS* is given by the following *eqs*(*CS*).

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

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

definition
  $items\text{-}of\ rhs\ X \equiv \{Trn\ X\ r \mid r.\ (Trn\ X\ r) \in rhs\}$

The following *rexp-of rhs X* combines all regular expressions in *X*-items using *ALT* to form a single regular expression. It will be used later to implement *arden-variate* and *rhs-subst*.

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

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

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

The following *rexp-of-lam rhs* combines pure regular expression items in *rhs* using *ALT* to form a single regular expression. When all variables inside *rhs* are eliminated, *rexp-of-lam rhs* is used to compute compute the regular expression corresponds to *rhs*.

definition
rexp-of-lam rhs ≡ *folds ALT NULL* (*the-r* ‘ *lam-of rhs*)

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

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

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

definition
append-rhs-rexp rhs rexp ≡ (*attach-rexp rexp*) ‘ *rhs*

With the help of the two functions immediately above, Ardens' transformation on right hand side *rhs* is implemented by the following function *arden-variate X rhs*. After this transformation, the recursive occurence of *X* in *rhs* will be eliminated, while the string set defined by *rhs* is kept unchanged.

definition
arden-variate X rhs ≡
append-rhs-rexp (*rhs* − *items-of rhs X*) (*STAR* (*rexp-of rhs X*))

Suppose the equation defining *X* is $X = xrhs$, the purpose of *rhs-subst* is to substitute all occurences of *X* in *rhs* by *xrhs*. A litte thought may reveal that the final result should be: first append $(a_1|a_2|\ldots|a_n)$ to every item of *xrhs* and then union the result with all non-*X*-items of *rhs*.

definition
rhs-subst rhs X xrhs ≡
(*rhs* − (*items-of rhs X*)) ∪ (*append-rhs-rexp xrhs* (*rexp-of rhs X*))

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

definition
eqs-subst ES X xrhs ≡ {(*Y*, *rhs-subst yrhs X xrhs*) | *Y yrhs*. (*Y*, *yrhs*) ∈ *ES*}

The computation of regular expressions for equivalence classes is accomplished using a iteration principle given by the following lemma.

lemma *wf-iter* [*rule-format*]:
fixes *f*
assumes *step*: ⋀ *e*. ⟦*P e*; ¬ *Q e*⟧ ⟹ (∃ *e′*. *P e′* ∧ (*f*(*e′*), *f*(*e*)) ∈ *less-than*)
shows *pe*: *P e* ⟶ (∃ *e′*. *P e′* ∧ *Q e′*)
proof(*induct e rule*: *wf-induct*

[*OF wf-inv-image*[*OF wf-less-than*, **where** $f = f$]], *clarify*)
fix x
assume h [*rule-format*]:
$\forall y.\ (y, x) \in$ *inv-image less-than* $f \longrightarrow P\ y \longrightarrow (\exists e'.\ P\ e' \wedge Q\ e')$
and *px*: $P\ x$
show $\exists e'.\ P\ e' \wedge Q\ e'$
proof(*cases* $Q\ x$)
assume $Q\ x$ **with** *px* **show** *?thesis* **by** *blast*
next
assume *nq*: $\neg\ Q\ x$
from *step* [*OF px nq*]
obtain e' **where** *pe'*: $P\ e'$ **and** *ltf*: $(f\ e', f\ x) \in$ *less-than* **by** *auto*
show *?thesis*
proof(*rule h*)
from *ltf* **show** $(e', x) \in$ *inv-image less-than* f
by (*simp add*:*inv-image-def*)
next
from *pe'* **show** $P\ e'$ **.**
qed
qed
qed

The P in lemma *wf-iter* is an invaiant kept throughout the iteration procedure. The particular invariant used to solve our problem is defined by function *Inv*(*ES*), an invariant over equal system *ES*. Every definition starting next till *Inv* stipulates a property to be satisfied by *ES*.

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

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

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

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

The following *rhs-nonempty rhs* requires regular expressions occuring in transitional items of *rhs* does not contain empty string. This is necessary for the application of Arden's transformation to *rhs*.

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

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

definition
ardenable ES $\equiv \forall\ X\ rhs.\ (X, rhs) \in ES \longrightarrow$ *rhs-nonempty rhs*

definition
  non-empty ES ≡ ∀ *X rhs*. (*X*, *rhs*) ∈ *ES* ⟶ *X* ≠ {}

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

definition
  finite-rhs ES ≡ ∀ *X rhs*. (*X*, *rhs*) ∈ *ES* ⟶ *finite rhs*

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

definition
  classes-of rhs ≡ {*X*. ∃ *r*. *Trn X r* ∈ *rhs*}

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

definition
  lefts-of ES ≡ {*Y* | *Y yrhs*. (*Y*, *yrhs*) ∈ *ES*}

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

definition
  self-contained ES ≡ ∀ (*X*, *xrhs*) ∈ *ES*. *classes-of xrhs* ⊆ *lefts-of ES*

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

definition
  Inv ES ≡ *valid-eqns ES* ∧ *finite ES* ∧ *distinct-equas ES* ∧ *ardenable ES* ∧
      non-empty ES ∧ *finite-rhs ES* ∧ *self-contained ES*

7.1 The proof of this direction

7.1.1 Basic properties

The following are some basic properties of the above definitions.

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

lemma *finite-snd-Trn*:
  assumes *finite*:*finite rhs*
  shows *finite* {r_2. *Trn Y* r_2 ∈ *rhs*} (**is** *finite ?B*)
proof−
  def *rhs′* ≡ {*e* ∈ *rhs*. ∃ *r*. *e* = *Trn Y r*}
  have *?B* = (*snd o the-Trn*) ‘ *rhs′* **using** *rhs′-def* **by** (*auto simp*:*image-def*)
  moreover have *finite rhs′* **using** *finite rhs′-def* **by** *auto*
  ultimately show *?thesis* **by** *simp*
qed

```
lemma rexp-of-empty:
  assumes finite:finite rhs
  and nonempty:rhs-nonempty rhs
  shows [] ∉ L (rexp-of rhs X)
using finite nonempty rhs-nonempty-def
by (drule-tac finite-snd-Trn[where Y = X], auto simp:rexp-of-def items-of-def)

lemma [intro!]:
  P (Trn X r) ⟹ (∃ a. (∃ r. a = Trn X r ∧ P a)) by auto

lemma finite-items-of:
  finite rhs ⟹ finite (items-of rhs X)
by (auto simp:items-of-def intro:finite-subset)

lemma lang-of-rexp-of:
  assumes finite:finite rhs
  shows L (items-of rhs X) = X ;; (L (rexp-of rhs X))
proof -
  have finite ((snd ∘ the-Trn) ` items-of rhs X) using finite-items-of[OF finite]
by auto
  thus ?thesis
    apply (auto simp:rexp-of-def Seq-def items-of-def)
    apply (rule-tac x = s1 in exI, rule-tac x = s2 in exI, auto)
    by (rule-tac x= Trn X r in exI, auto simp:Seq-def)
qed

lemma rexp-of-lam-eq-lam-set:
  assumes finite: finite rhs
  shows L (rexp-of-lam rhs) = L (lam-of rhs)
proof -
  have finite (the-r ` {Lam r |r. Lam r ∈ rhs}) using finite
    by (rule-tac finite-imageI, auto intro:finite-subset)
  thus ?thesis by (auto simp:rexp-of-lam-def lam-of-def)
qed

lemma [simp]:
  L (attach-rexp r xb) = L xb ;; L r
apply (cases xb, auto simp:Seq-def)
apply(rule-tac x = s1 @ s1' in exI, rule-tac x = s2' in exI)
apply(auto simp: Seq-def)
done

lemma lang-of-append-rhs:
  L (append-rhs-rexp rhs r) = L rhs ;; L r
apply (auto simp:append-rhs-rexp-def image-def)
apply (auto simp:Seq-def)
apply (rule-tac x = L xb ;; L r in exI, auto simp add:Seq-def)
by (rule-tac x = attach-rexp r xb in exI, auto simp:Seq-def)
```

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

lemma *lefts-of-union-distrib*:
  lefts-of $A \cup$ *lefts-of* $B =$ *lefts-of* $(A \cup B)$
by (*auto simp*:*lefts-of-def*)

7.1.2 Intialization

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

lemma *defined-by-str*:
  $[\![s \in X; X \in UNIV\ //\ (\approx Lang)]\!] \Longrightarrow X = (\approx Lang)\ ``\ \{s\}$
by (*auto simp*:*quotient-def Image-def str-eq-rel-def*)

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

lemma *l-eq-r-in-eqs*:
  assumes *X-in-eqs*: $(X, xrhs) \in (eqs\ (UNIV\ //\ (\approx Lang)))$
  shows $X = L\ xrhs$
proof
  show $X \subseteq L\ xrhs$
  proof
    fix x
    assume (*1*): $x \in X$
    show $x \in L\ xrhs$
    proof (*cases* $x = []$)
      assume *empty*: $x = []$

```
      thus ?thesis using X-in-eqs (1)
        by (auto simp:eqs-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 // (≈Lang) using X-in-eqs by (auto simp:eqs-def)
      then obtain Y
        where Y ∈ UNIV // (≈Lang)
        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 // (≈Lang) ∧ 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: eqs-def init-rhs-def)
    qed
  qed
next
  show L xrhs ⊆ X using X-in-eqs
    by (auto simp:eqs-def init-rhs-def transition-def)
qed

lemma finite-init-rhs:
  assumes finite: finite CS
  shows finite (init-rhs CS X)
proof−
  have finite {Trn Y (CHAR c) |Y c. Y ∈ CS ∧ Y ;; {[c]} ⊆ X} (is finite ?A)
  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
    hence finite S using S-def
      by (rule-tac B = CS × UNIV in finite-subset, auto)
    moreover have ?A = h ‘ S by (auto simp: S-def h-def image-def)
    ultimately show ?thesis
      by auto
  qed
  thus ?thesis by (simp add:init-rhs-def transition-def)
qed

lemma init-ES-satisfy-Inv:
  assumes finite-CS: finite (UNIV // (≈Lang))
  shows Inv (eqs (UNIV // (≈Lang)))
proof −
  have finite (eqs (UNIV // (≈Lang))) using finite-CS
```

```
    by (simp add:eqs-def)
  moreover have distinct-equas (eqs (UNIV // (≈Lang)))
    by (simp add:distinct-equas-def eqs-def)
  moreover have ardenable (eqs (UNIV // (≈Lang)))
   by (auto simp add:ardenable-def eqs-def init-rhs-def rhs-nonempty-def del:L-rhs.simps)
  moreover have valid-eqns (eqs (UNIV // (≈Lang)))
    using l-eq-r-in-eqs by (simp add:valid-eqns-def)
  moreover have non-empty (eqs (UNIV // (≈Lang)))
    by (auto simp:non-empty-def eqs-def quotient-def Image-def str-eq-rel-def)
  moreover have finite-rhs (eqs (UNIV // (≈Lang)))
    using finite-init-rhs[OF finite-CS]
    by (auto simp:finite-rhs-def eqs-def)
  moreover have self-contained (eqs (UNIV // (≈Lang)))
    by (auto simp:self-contained-def eqs-def init-rhs-def classes-of-def lefts-of-def)
  ultimately show ?thesis by (simp add:Inv-def)
qed
```

7.1.3 Interation step

From this point until *iteration-step*, it is proved that there exists iteration steps which keep *Inv*(*ES*) while decreasing the size of *ES*.

```
lemma arden-variate-keeps-eq:
  assumes l-eq-r: X = L rhs
  and not-empty: [] ∉ L (rexp-of rhs X)
  and finite: finite rhs
  shows X = L (arden-variate X rhs)
proof −
  def A ≡ L (rexp-of rhs X)
  def b ≡ rhs − items-of rhs X
  def B ≡ L b
  have X = B ;; A⋆
  proof−
    have rhs = items-of rhs X ∪ b by (auto simp:b-def items-of-def)
    hence L rhs = L(items-of rhs X ∪ b) by simp
   hence L rhs = L(items-of rhs X) ∪ B by (simp only:L-rhs-union-distrib B-def)
    with lang-of-rexp-of
    have L rhs = X ;; A ∪ B  using finite by (simp only:B-def b-def A-def)
    thus ?thesis
      using l-eq-r not-empty
      apply (drule-tac B = B and X = X in ardens-revised)
      by (auto simp:A-def simp del:L-rhs.simps)
  qed
  moreover have L (arden-variate X rhs) = (B ;; A⋆) (is ?L = ?R)
    by (simp only:arden-variate-def L-rhs-union-distrib lang-of-append-rhs
                  B-def A-def b-def L-rexp.simps seq-union-distrib-left)
   ultimately show ?thesis by simp
qed

lemma append-keeps-finite:
```

finite rhs $\Longrightarrow$ *finite* (*append-rhs-rexp rhs r*)
by (*auto simp*:*append-rhs-rexp-def*)

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

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

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

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

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

lemma *rhs-subst-keeps-nonempty*:
$[\![$*rhs-nonempty rhs*; *rhs-nonempty xrhs*$]\!]$ $\Longrightarrow$ *rhs-nonempty* (*rhs-subst rhs X xrhs*)
by (*simp only*:*rhs-subst-def append-keeps-nonempty nonempty-set-union nonempty-set-sub*)

lemma *rhs-subst-keeps-eq*:
assumes *substor*: $X = L$ *xrhs*
and *finite*: *finite rhs*
shows *L* (*rhs-subst rhs X xrhs*) = *L rhs* (**is** *?Left* = *?Right*)
proof−
def $A \equiv L$ (*rhs* $-$ *items-of rhs X*)
have *?Left* = $A \cup L$ (*append-rhs-rexp xrhs* (*rexp-of rhs X*))
by (*simp only*:*rhs-subst-def L-rhs-union-distrib A-def*)
moreover have *?Right* = $A \cup L$ (*items-of rhs X*)
proof−
have *rhs* = (*rhs* $-$ *items-of rhs X*) $\cup$ (*items-of rhs X*) **by** (*auto simp*:*items-of-def*)
thus *?thesis* **by** (*simp only*:*L-rhs-union-distrib A-def*)
qed
moreover have *L* (*append-rhs-rexp xrhs* (*rexp-of rhs X*)) = *L* (*items-of rhs X*)
using *finite substor* **by** (*simp only*:*lang-of-append-rhs lang-of-rexp-of*)
ultimately show *?thesis* **by** *simp*
qed

lemma *rhs-subst-keeps-finite-rhs*:
$[\![$*finite rhs*; *finite yrhs*$]\!]$ $\Longrightarrow$ *finite* (*rhs-subst rhs Y yrhs*)

```
by (auto simp:rhs-subst-def append-keeps-finite)

lemma eqs-subst-keeps-finite:
  assumes finite:finite (ES:: (string set × rhs-item set) set)
  shows finite (eqs-subst ES Y yrhs)
proof −
  have finite {(Ya, rhs-subst yrhsa Y yrhs) |Ya yrhsa. (Ya, yrhsa) ∈ ES}
                                                   (is finite ?A)
  proof−
    def eqns' ≡ {((Ya::string set), yrhsa)| Ya yrhsa. (Ya, yrhsa) ∈ ES}
    def h ≡ λ ((Ya::string set), yrhsa). (Ya, rhs-subst yrhsa Y yrhs)
    have finite (h ‘ eqns') using finite h-def eqns'-def by auto
    moreover have ?A = h ‘ eqns' by (auto simp:h-def eqns'-def)
    ultimately show ?thesis by auto
  qed
  thus ?thesis by (simp add:eqs-subst-def)
qed

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

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

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

lemma lefts-of-keeps-cls:
  lefts-of (eqs-subst ES Y yrhs) = lefts-of ES
by (auto simp:lefts-of-def eqs-subst-def)

lemma rhs-subst-updates-cls:
  X ∉ classes-of xrhs ⟹
      classes-of (rhs-subst rhs X xrhs) = classes-of rhs ∪ classes-of xrhs − {X}
apply (simp only:rhs-subst-def append-rhs-keeps-cls
                          classes-of-union-distrib[THEN sym])
by (auto simp:classes-of-def items-of-def)

lemma eqs-subst-keeps-self-contained:
  fixes Y
  assumes sc: self-contained (ES ∪ {(Y, yrhs)}) (is self-contained ?A)
  shows self-contained (eqs-subst ES Y (arden-variate Y yrhs))
                                          (is self-contained ?B)
```

```
proof−
  { fix X xrhs'
    assume (X, xrhs') ∈ ?B
    then obtain xrhs
      where xrhs-xrhs': xrhs' = rhs-subst xrhs Y (arden-variate Y yrhs)
      and X-in: (X, xrhs) ∈ ES by (simp add:eqs-subst-def, blast)
    have classes-of xrhs' ⊆ lefts-of ?B
    proof−
      have lefts-of ?B = lefts-of ES by (auto simp add:lefts-of-def eqs-subst-def)
      moreover have classes-of xrhs' ⊆ lefts-of ES
      proof−
        have classes-of xrhs' ⊆
                    classes-of xrhs ∪ classes-of (arden-variate Y yrhs) − {Y}
        proof−
          have Y ∉ classes-of (arden-variate Y yrhs)
            using arden-variate-removes-cl by simp
          thus ?thesis using xrhs-xrhs' by (auto simp:rhs-subst-updates-cls)
        qed
        moreover have classes-of xrhs ⊆ lefts-of ES ∪ {Y} using X-in sc
          apply (simp only:self-contained-def lefts-of-union-distrib[THEN sym])
          by (drule-tac x = (X, xrhs) in bspec, auto simp:lefts-of-def)
        moreover have classes-of (arden-variate Y yrhs) ⊆ lefts-of ES ∪ {Y}
          using sc
          by (auto simp add:arden-variate-removes-cl self-contained-def lefts-of-def)
        ultimately show ?thesis by auto
      qed
      ultimately show ?thesis by simp
    qed
  } thus ?thesis by (auto simp only:eqs-subst-def self-contained-def)
qed

lemma eqs-subst-satisfy-Inv:
  assumes Inv-ES: Inv (ES ∪ {(Y, yrhs)})
  shows Inv (eqs-subst ES Y (arden-variate Y yrhs))
proof −
  have finite-yrhs: finite yrhs
    using Inv-ES by (auto simp:Inv-def finite-rhs-def)
  have nonempty-yrhs: rhs-nonempty yrhs
    using Inv-ES by (auto simp:Inv-def ardenable-def)
  have Y-eq-yrhs: Y = L yrhs
    using Inv-ES by (simp only:Inv-def valid-eqns-def, blast)
  have distinct-equas (eqs-subst ES Y (arden-variate Y yrhs))
    using Inv-ES
    by (auto simp:distinct-equas-def eqs-subst-def Inv-def)
  moreover have finite (eqs-subst ES Y (arden-variate Y yrhs))
    using Inv-ES by (simp add:Inv-def eqs-subst-keeps-finite)
  moreover have finite-rhs (eqs-subst ES Y (arden-variate Y yrhs))
  proof−
    have finite-rhs ES using Inv-ES
```

```
      by (simp add:Inv-def finite-rhs-def)
    moreover have finite (arden-variate Y yrhs)
    proof -
      have finite yrhs using Inv-ES
        by (auto simp:Inv-def finite-rhs-def)
      thus ?thesis using arden-variate-keeps-finite by simp
    qed
    ultimately show ?thesis
      by (simp add:eqs-subst-keeps-finite-rhs)
  qed
  moreover have ardenable (eqs-subst ES Y (arden-variate Y yrhs))
  proof -
    { fix X rhs
      assume (X, rhs) ∈ ES
      hence rhs-nonempty rhs  using prems Inv-ES
        by (simp add:Inv-def ardenable-def)
      with nonempty-yrhs
      have rhs-nonempty (rhs-subst rhs Y (arden-variate Y yrhs))
        by (simp add:nonempty-yrhs
              rhs-subst-keeps-nonempty arden-variate-keeps-nonempty)
    } thus ?thesis by (auto simp add:ardenable-def eqs-subst-def)
  qed
  moreover have valid-eqns (eqs-subst ES Y (arden-variate Y yrhs))
  proof−
    have Y = L (arden-variate Y yrhs)
      using Y-eq-yrhs Inv-ES finite-yrhs nonempty-yrhs
      by (rule-tac arden-variate-keeps-eq, (simp add:rexp-of-empty)+)
    thus ?thesis using Inv-ES
      by (clarsimp simp add:valid-eqns-def
             eqs-subst-def rhs-subst-keeps-eq Inv-def finite-rhs-def
                 simp del:L-rhs.simps)
  qed
  moreover have
    non-empty-subst: non-empty (eqs-subst ES Y (arden-variate Y yrhs))
    using Inv-ES by (auto simp:Inv-def non-empty-def eqs-subst-def)
  moreover
  have self-subst: self-contained (eqs-subst ES Y (arden-variate Y yrhs))
    using Inv-ES eqs-subst-keeps-self-contained by (simp add:Inv-def)
  ultimately show ?thesis using Inv-ES by (simp add:Inv-def)
qed

lemma eqs-subst-card-le:
  assumes finite: finite (ES::(string set × rhs-item set) set)
  shows card (eqs-subst ES Y yrhs) <= card ES
proof−
  def f ≡ λ x. ((fst x)::string set, rhs-subst (snd x) Y yrhs)
  have eqs-subst ES Y yrhs = f ` ES
    apply (auto simp:eqs-subst-def f-def image-def)
    by (rule-tac x = (Ya, yrhsa) in bexI, simp+)
```

```
  thus ?thesis using finite by (auto intro:card-image-le)
qed

lemma eqs-subst-cls-remains:
  (X, xrhs) ∈ ES ⟹ ∃ xrhs'. (X, xrhs') ∈ (eqs-subst ES Y yrhs)
by (auto simp:eqs-subst-def)

lemma card-noteq-1-has-more:
  assumes card:card S ≠ 1
  and e-in: e ∈ S
  and finite: finite S
  obtains e' where e' ∈ S ∧ e ≠ e'
proof−
  have card (S − {e}) > 0
  proof −
    have card S > 1 using card e-in finite
      by (case-tac card S, auto)
    thus ?thesis using finite e-in by auto
  qed
  hence S − {e} ≠ {} using finite by (rule-tac notI, simp)
  thus (⋀e'. e' ∈ S ∧ e ≠ e' ⟹ thesis) ⟹ thesis by auto
qed

lemma iteration-step:
  assumes Inv-ES: Inv ES
  and     X-in-ES: (X, xrhs) ∈ ES
  and     not-T: card ES ≠ 1
  shows ∃ ES'. (Inv ES' ∧ (∃ xrhs'.(X, xrhs') ∈ ES')) ∧
              (card ES', card ES) ∈ less-than (is ∃ ES'. ?P ES')
proof −
  have finite-ES: finite ES using Inv-ES by (simp add:Inv-def)
  then obtain Y yrhs
    where Y-in-ES: (Y, yrhs) ∈ ES and not-eq: (X, xrhs) ≠ (Y, yrhs)
    using not-T X-in-ES by (drule-tac card-noteq-1-has-more, auto)
  def ES' == ES − {(Y, yrhs)}
  let ?ES'' = eqs-subst ES' Y (arden-variate Y yrhs)
  have ?P ?ES''
  proof −
    have Inv ?ES'' using Y-in-ES Inv-ES
      by (rule-tac eqs-subst-satisfy-Inv, simp add:ES'-def insert-absorb)
    moreover have ∃xrhs'. (X, xrhs') ∈ ?ES''  using not-eq X-in-ES
      by (rule-tac ES = ES' in eqs-subst-cls-remains, auto simp add:ES'-def)
    moreover have (card ?ES'', card ES) ∈ less-than
    proof −
      have finite ES' using finite-ES ES'-def by auto
      moreover have card ES' < card ES using finite-ES Y-in-ES
        by (auto simp:ES'-def card-gt-0-iff intro:diff-Suc-less)
      ultimately show ?thesis
        by (auto dest:eqs-subst-card-le elim:le-less-trans)
```

```
    qed
    ultimately show ?thesis by simp
  qed
  thus ?thesis by blast
qed
```

7.1.4 Conclusion of the proof

From this point until *hard-direction*, the hard direction is proved through a simple application of the iteration principle.

```
lemma iteration-conc:
  assumes history: Inv ES
  and     X-in-ES: ∃ xrhs. (X, xrhs) ∈ ES
  shows
  ∃ ES'. (Inv ES' ∧ (∃ xrhs'. (X, xrhs') ∈ ES')) ∧ card ES' = 1
                                                        (is ∃ ES'. ?P ES')
proof (cases card ES = 1)
  case True
  thus ?thesis using history X-in-ES
    by blast
next
  case False
  thus ?thesis using history iteration-step X-in-ES
    by (rule-tac f = card in wf-iter, auto)
qed

lemma last-cl-exists-rexp:
  assumes ES-single: ES = {(X, xrhs)}
  and Inv-ES: Inv ES
  shows ∃ (r::rexp). L r = X (is ∃ r. ?P r)
proof−
  let ?A = arden-variate X xrhs
  have ?P (rexp-of-lam ?A)
  proof −
    have L (rexp-of-lam ?A) = L (lam-of ?A)
    proof(rule rexp-of-lam-eq-lam-set)
      show finite (arden-variate X xrhs) using Inv-ES ES-single
        by (rule-tac arden-variate-keeps-finite,
                       auto simp add:Inv-def finite-rhs-def)
    qed
    also have . . . = L ?A
    proof−
      have lam-of ?A = ?A
      proof−
        have classes-of ?A = {} using Inv-ES ES-single
          by (simp add:arden-variate-removes-cl
                       self-contained-def Inv-def lefts-of-def)
        thus ?thesis
          by (auto simp only:lam-of-def classes-of-def, case-tac x, auto)
```

```
      qed
      thus ?thesis by simp
    qed
    also have ... = X
    proof(rule arden-variate-keeps-eq [THEN sym])
      show X = L xrhs using Inv-ES ES-single
        by (auto simp only:Inv-def valid-eqns-def)
    next
      from Inv-ES ES-single show [] ∉ L (rexp-of xrhs X)
        by(simp add:Inv-def ardenable-def rexp-of-empty finite-rhs-def)
    next
      from Inv-ES ES-single show finite xrhs
        by (simp add:Inv-def finite-rhs-def)
    qed
    finally show ?thesis by simp
  qed
  thus ?thesis by auto
qed

lemma every-eqcl-has-reg:
  assumes finite-CS: finite (UNIV // (≈Lang))
  and X-in-CS: X ∈ (UNIV // (≈Lang))
  shows ∃ (reg::rexp). L reg = X (is ∃ r. ?E r)
proof −
  from X-in-CS have ∃ xrhs. (X, xrhs) ∈ (eqs (UNIV // (≈Lang)))
    by (auto simp:eqs-def init-rhs-def)
  then obtain ES xrhs where Inv-ES: Inv ES
    and X-in-ES: (X, xrhs) ∈ ES
    and card-ES: card ES = 1
    using finite-CS X-in-CS init-ES-satisfy-Inv iteration-conc
    by blast
  hence ES-single-equa: ES = {(X, xrhs)}
    by (auto simp:Inv-def dest!:card-Suc-Diff1 simp:card-eq-0-iff)
  thus ?thesis using Inv-ES
    by (rule last-cl-exists-rexp)
qed

lemma finals-in-partitions:
  shows finals A ⊆ (UNIV // ≈A)
unfolding finals-def
unfolding quotient-def
by auto

theorem hard-direction:
  assumes finite-CS: finite (UNIV // ≈A)
  shows ∃ r::rexp. A = L r
proof −
  have ∀ X ∈ (UNIV // ≈A). ∃ reg::rexp. X = L reg
    using finite-CS every-eqcl-has-reg by blast
```

then obtain *f*
 where *f-prop*: ∀ *X* ∈ (*UNIV* // ≈*A*). *X* = *L* ((*f X*)::*rexp*)
 by (*auto dest*: *bchoice*)
def *rs* ≡ *f* ‘ (*finals A*)
have *A* = ⋃ (*finals A*) **using** *lang-is-union-of-finals* **by** *auto*
also have . . . = *L* (*folds ALT NULL rs*)
proof −
 have *finite rs*
 proof −
  have *finite* (*finals A*)
   using *finite-CS finals-in-partitions*[*of A*]
   by (*erule-tac finite-subset*, *simp*)
  thus *?thesis* **using** *rs-def* **by** *auto*
 qed
 thus *?thesis*
  using *f-prop rs-def finals-in-partitions*[*of A*] **by** *auto*
qed
finally show *?thesis* **by** *blast*
qed

end
theory *Myhill-2*
 imports *Myhill-1*
begin

8 Direction *regular language ⇒finite partition*

8.1 The scheme

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

definition
 str-eq :: *string* ⇒ *lang* ⇒ *string* ⇒ *bool* (- ≈- -)
where
 x ≈*Lang y* ≡ (*x*, *y*) ∈ (≈*Lang*)

The main lemma (*rexp-imp-finite*) is proved by a structural induction over regular expressions. While base cases (cases for *NULL*, *EMPTY*, *CHAR*) are quite straight forward, the inductive cases are rather involved. What we have when starting to prove these inductive caes is that the partitions induced by the componet language are finite. The basic idea to show the finiteness of the partition induced by the composite language is to attach a tag $tag(x)$ to every string x. The tags are made of equivalent classes from the component partitions. Let *tag* be the tagging function and *Lang* be the composite language, it can be 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
  $(=f=) = \{(x, y) \mid x\ y.\ f\ x = f\ y\}$

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*: *finite* (*range f*) ⟹ *finite* (*f* ‘ *A*)
 by (*rule-tac* $B = \{y.\ \exists x.\ y = f\ x\}$ **in** *finite-subset*, *auto simp*:*image-def*)

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*:

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

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
```

8.2 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.

8.2.1 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
```

8.2.2 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*)

8.2.3 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* // ($\approx$\{[*c*]\}))
by (*rule finite-subset*[*OF quot-char-subset*]) (*simp*)

8.2.4 The inductive case for *ALT*

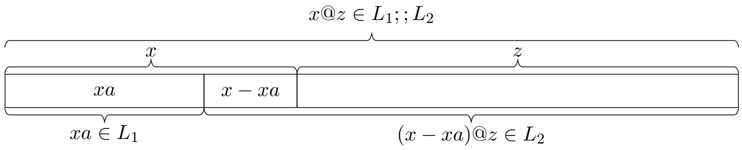
definition
  tag-str-ALT :: *lang* $\Rightarrow$ *lang* $\Rightarrow$ *string* $\Rightarrow$ (*lang* $\times$ *lang*)
where
  tag-str-ALT L1 L2 = (λ*x*. ($\approx$*L1* `` \{*x*\}, $\approx$*L2* `` \{*x*\}))

lemma *quot-union-finiteI* [*intro*]:
  fixes *L1 L2*::*lang*
  assumes *finite1*: *finite* (*UNIV* // $\approx$*L1*)
  and *finite2*: *finite* (*UNIV* // $\approx$*L2*)
  shows *finite* (*UNIV* // $\approx$(*L1* $\cup$ *L2*))
proof (*rule-tac tag* = *tag-str-ALT L1 L2* **in** *tag-finite-imageD*)
  show $\bigwedge$*x y*. *tag-str-ALT L1 L2 x* = *tag-str-ALT L1 L2 y* $\Longrightarrow$ *x* $\approx$(*L1* $\cup$ *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* // $\approx$*L1*) $\times$ (*UNIV* // $\approx$*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

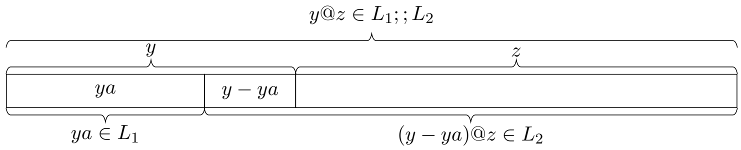
8.2.5 The inductive case for *SEQ*

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.

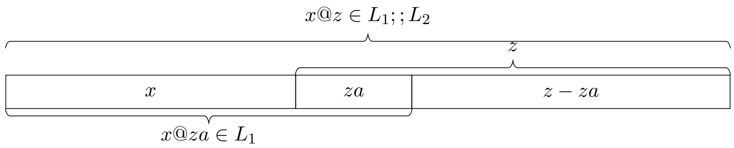
definition
  tag-str-SEQ :: *lang* $\Rightarrow$ *lang* $\Rightarrow$ *string* $\Rightarrow$ (*lang* $\times$ *lang set*)



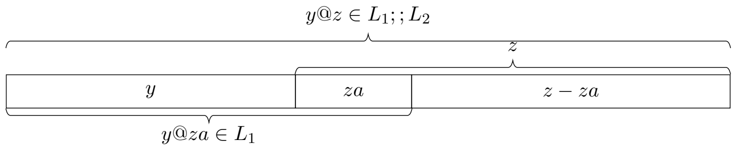
(a) First possible way to split $x@z$



(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

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* $\vee$ *?Split2*) **.**
moreover have *?Split1* $\Longrightarrow \exists\ ya \le y.\ (x\ @\ ya) \in L_1 \wedge (y - ya) \in L_2$
using *in-seq* **by** (*rule-tac* $x = s_1 - x$ **in** *exI*, *auto elim:prefixE*)
moreover have *?Split2* $\Longrightarrow \exists\ xa \le x.\ xa \in L_1 \wedge (x - xa)\ @\ y \in L_2$
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 $(\exists\ xa \le x.\ xa \in L_1 \wedge (x - xa)\ @\ z \in L_2) \vee$
$(\exists\ za \le z.\ (x\ @\ za) \in L_1 \wedge (z - za) \in L_2)$ **.**
— It can be shown that *?thesis* holds in either case:
moreover {
— The case for the first split:
fix *xa*
assume *h1*: $xa \le x$ **and** *h2*: $xa \in L_1$ **and** *h3*: $(x - xa)\ @\ z \in L_2$
— The following subgoal implements the structure transfer:
obtain *ya*
where $ya \le y$
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 $\{\approx L_2$ “ $\{x - xa\}\ |xa.\ xa \le x \wedge xa \in L_1\} =$
$\{\approx L_2$ “ $\{y - ya\}\ |ya.\ ya \le y \wedge ya \in L_1\}$ (**is** *?Left* = *?Right*)
using *tag-xy* **unfolding** *tag-str-SEQ-def* **by** *simp*
— Since $xa \le x$ and $xa \in L_1$ hold, it is not difficult to show:
moreover have $\approx L_2$ “ $\{x - xa\} \in$ *?Left* **using** *h1 h2* **by** *auto*
— Through tag equality, equivalent class $\approx L_2$ “ $\{x - xa\}$ also belongs to the *?Right*:
ultimately have $\approx L_2$ “ $\{x - xa\} \in$ *?Right* **by** *simp*
— From this, the counterpart of *xa* in *y* is obtained:
then obtain *ya*

where *eq-xya*: $\approx L_2$ `` $\{x - xa\} = \approx L_2$ `` $\{y - ya\}$
and *pref-ya*: $ya \leq y$ **and** *ya-in*: $ya \in L_1$
by *simp blast*
— It can be proved that *ya* has the desired property:
have $(y - ya)@z \in L_2$
proof −
from *eq-xya* **have** $(x - xa) \approx L_2 (y - ya)$
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*, $y @ z \in L_1 ;; L_2$ is derived easily.
hence $y @ z \in L_1 ;; L_2$ **by** (*erule-tac prefixE*, *auto simp*:*Seq-def*)
} **moreover** **{**
— The other case is even more simpler:
fix *za*
assume *h1*: $za \leq z$ **and** *h2*: $(x @ za) \in L_1$ **and** *h3*: $z - za \in L_2$
have $y @ za \in L_1$
proof−
have $\approx L_1$ `` $\{x\} = \approx L_1$ `` $\{y\}$
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** $y @ z \in L_1 ;; L_2$
by (*drule-tac* $A = L_1$ **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 $\bigwedge x\ y.$ *tag-str-SEQ L1 L2 x* = *tag-str-SEQ L1 L2 y* $\Longrightarrow$ *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

8.2.6 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:

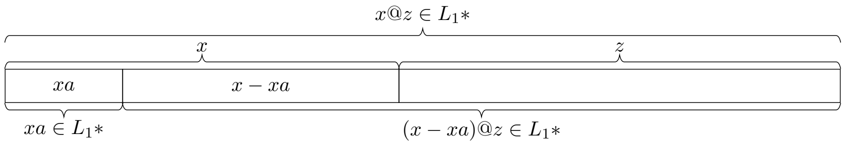
   (a) A y-prefix ya corresponding to xa can be found, which satisfies conditions: $ya \in L_1*$ and $(y - ya)@za \in L_1$.

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

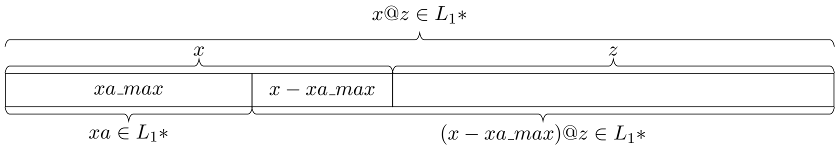
   (c) With fact $ya \in L_1*$, we finally get $y@z \in L_1*$.

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.

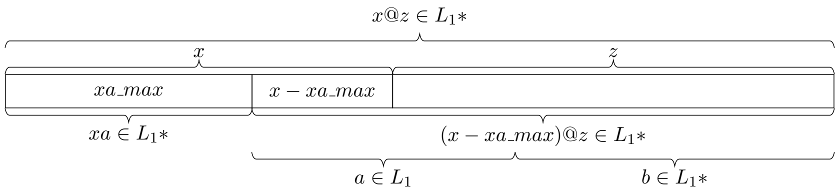
definition



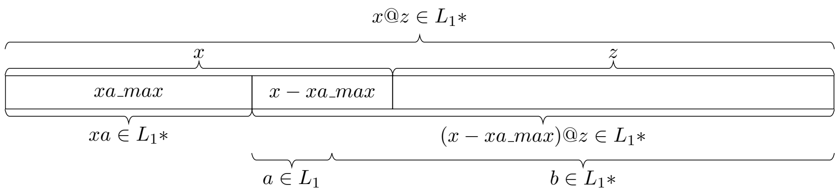
(a) First split



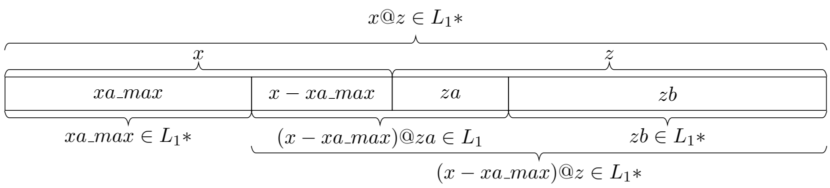
(b) Max split



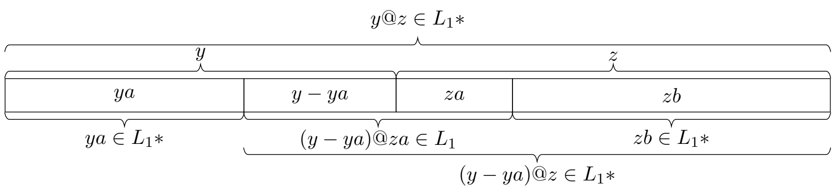
(c) Max split with a and b (the right situation)



(d) Max split with a and b (the wrong situation)



(e) Last split



(f) Structure transferred to y

Figure 2: The case for $STAR$

```
  tag-str-STAR :: lang ⇒ string ⇒ lang set
where
  tag-str-STAR L1 = (λx. {≈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:
  fixes v w
  assumes eq-tag: tag-str-STAR L₁ v = tag-str-STAR L₁ w
  shows (v::string) ≈(L₁⋆) 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 ∈ L₁⋆
      and tag-xy: tag-str-STAR L₁ x = tag-str-STAR L₁ y
    have y @ z ∈ L₁⋆
```

proof(*cases* $x = []$)
— The degenerated case when x is a null string is easy to prove:
case *True*
with *tag-xy* **have** $y = []$
by (*auto simp 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 $?S = \{xa.\ xa < x \wedge xa \in L_1\star \wedge (x - xa)\ @\ z \in L_1\star\}$
have *finite ?S*
by (*rule-tac* $B = \{xa.\ xa < x\}$ **in** *finite-subset*,
auto simp:*finite-strict-prefix-set*)
moreover have $?S \neq \{\}$ **using** *False xz-in-star*
by (*simp*, *rule-tac* $x = []$ **in** *exI*, *auto simp*:*strict-prefix-def*)
— Since *?S* is finite, we can always single out the longest and name it *xa-max*:
ultimately have $\exists$ *xa-max* $\in$ *?S*. $\forall$ *xa* $\in$ *?S*. *length xa* $\leq$ *length xa-max*
using *finite-set-has-max* **by** *blast*
then obtain *xa-max*
where *h1*: *xa-max* $< x$
and *h2*: *xa-max* $\in L_1\star$
and *h3*: $(x - \textit{xa-max})$ @ $z \in L_1\star$
and *h4*:$\forall$ $xa < x$. $xa \in L_1\star \wedge (x - xa)$ @ $z \in L_1\star$
$\longrightarrow$ *length xa* $\leq$ *length xa-max*
by *blast*
— By the equality of tags, the counterpart of *xa-max* among *y*-prefixes, named *ya*, can be found:
obtain *ya*
where *h5*: $ya < y$ **and** *h6*: $ya \in L_1\star$
and *eq-xya*: $(x - \textit{xa-max}) \approx L_1\ (y - ya)$
proof−
from *tag-xy* **have** $\{\approx L_1$ ‘‘ $\{x - xa\}\ |xa.\ xa < x \wedge xa \in L_1\star\} =$
$\{\approx L_1$ ‘‘ $\{y - xa\}\ |xa.\ xa < y \wedge xa \in L_1\star\}$ (**is** *?left* = *?right*)
by (*auto simp*:*tag-str-STAR-def*)
moreover have $\approx L_1$ ‘‘ $\{x - \textit{xa-max}\} \in$ *?left* **using** *h1 h2* **by** *auto*
ultimately have $\approx L_1$ ‘‘ $\{x - \textit{xa-max}\} \in$ *?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-max < x, x can be splitted into a and b such that:
    from h1 have (x - xa-max) @ z ≠ []
      by (auto simp:strict-prefix-def elim:prefixE)
    from star-decom [OF h3 this]
    obtain a b where a-in: a ∈ L₁
      and a-neq: a ≠ [] and b-in: b ∈ L₁⋆
      and ab-max: (x - xa-max) @ z = a @ b by blast
    — Now the candiates for za and zb are found:
    let ?za = a - (x - xa-max) and ?zb = b
    have pfx: (x - xa-max) ≤ a (is ?P1)
      and eq-z: z = ?za @ ?zb (is ?P2)
    proof -
      — Since (x - xa-max) @ z = a @ b, string (x - xa-max) @ z can
        be splitted in two ways:
      have ((x - xa-max) ≤ a ∧ (a - (x - xa-max)) @ b = z) ∨
        (a < (x - xa-max) ∧ ((x - xa-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-max)
          and b-eqs: ((x - xa-max) - a) @ z = b
        have False
        proof -
          let ?xa-max' = xa-max @ a
          have ?xa-max' < x
            using np h1 by (clarsimp simp:strict-prefix-def diff-prefix)
          moreover have ?xa-max' ∈ L₁⋆
            using a-in h2 by (simp add:star-intro3)
          moreover have (x - ?xa-max') @ z ∈ L₁⋆
            using b-eqs b-in np h1 by (simp add:diff-diff-appd)
          moreover have ¬ (length ?xa-max' ≤ 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-max)@?za ∈ L₁ using a-in by (auto elim:prefixE)
    — Now candidates ?za and ?zb have all the requred properteis.
    with eq-xya have (y - ya) @ ?za ∈ L₁
      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 ∈ L₁⋆ 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*) ≈(L_1⋆) *w*
proof−
— According to the definition of ≈*Lang*, proving *v* ≈(L_1⋆) *w* amounts to showing: for any string *u*, if *v* @ *u* ∈ (L_1⋆) then *w* @ *u* ∈ (L_1⋆) and vice versa. The reasoning pattern for both directions are the same, as derived in the following:
{ fix *x y z*
assume *xz-in-star*: *x* @ *z* ∈ L_1⋆
  and *tag-xy*: *tag-str-STAR* L_1 *x* = *tag-str-STAR* L_1 *y*
have *y* @ *z* ∈ L_1⋆
proof(*cases x* = [])
— The degenerated case when *x* is a null string is easy to prove:
case *True*
with *tag-xy* **have** *y* = []
  by (*auto simp*:*tag-str-STAR-def strict-prefix-def*)
thus *?thesis* **using** *xz-in-star True* **by** *simp*
next
— The case when *x* is not null, and *x* @ *z* is in L_1⋆,
case *False*
obtain *x-max*
  where *h1*: *x-max* < *x*
  and *h2*: *x-max* ∈ L_1⋆
  and *h3*: (*x* − *x-max*) @ *z* ∈ L_1⋆
  and *h4*:∀ *xa* < *x*. *xa* ∈ L_1⋆ ∧ (*x* − *xa*) @ *z* ∈ L_1⋆
    ⟶ *length xa* ≤ *length x-max*
proof−
  let *?S* = {*xa*. *xa* < *x* ∧ *xa* ∈ L_1⋆ ∧ (*x* − *xa*) @ *z* ∈ L_1⋆}
  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 ∈ L1
    by (auto simp:str-eq-def str-eq-rel-def)
  with b-in have ((y − ya) @ za) @ b ∈ L1⋆ 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 ⋀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
```

8.2.7 The conclusion

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

end

theory Myhill
 imports Myhill-2
begin
```

9 Direction *regular language ⇒finite partition*

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:

$$
\begin{aligned}
\hat{\delta}_M([]) &\equiv s_M \\
\hat{\delta}_M(xa) &\equiv \delta_M(\hat{\delta}_M(x), a)
\end{aligned}
\tag{2}
$$

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:

$$
L(M) \equiv \{x \mid \hat{\delta}_M(x) \in F_M\} \tag{3}
$$

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:

$$
M_p \;\equiv\; (Q_M, \Sigma_M, \delta_M, p, F_M) \tag{4}
$$

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

$$
L(M_p) = L(M_q) \tag{5}
$$

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:

$$
x \approx_M y \;\;\equiv\;\; \hat{\delta}_M(x) \approx_M \hat{\delta}_M(y) \tag{6}
$$

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. Now, we get one direction of Myhill-Nerode Theorem:

Lemma 1 (Myhill-Nerode Theorem, Direction one)**.** *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 two)**.** *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).*

To prove this lemma, a DFA $M_{\mathcal{L}}$ is constructed out of $\approx_{\mathcal{L}}$ with:

$$Q_{M_{\mathcal{L}}} \equiv \{[\![x]\!]_{\approx_{\mathcal{L}}} \mid x \in \Sigma^*\} \tag{7a}$$
$$\Sigma_{M_{\mathcal{L}}} \equiv \Sigma_M \tag{7b}$$
$$\delta_{M_{\mathcal{L}}} \equiv (\lambda([\![x]\!]_{\approx_{\mathcal{L}}}, a).[\![xa]\!]_{\approx_{\mathcal{L}}}) \tag{7c}$$
$$s_{M_{\mathcal{L}}} \equiv [\![[]]\!]_{\approx_{\mathcal{L}}} \tag{7d}$$
$$F_{M_{\mathcal{L}}} \equiv \{[\![x]\!]_{\approx_{\mathcal{L}}} \mid x \in \mathcal{L}\} \tag{7e}$$

From the assumption of lemma 2, we have that $\{[\![x]\!]_{\approx_{\mathcal{L}}} \mid x \in \Sigma^*\}$ is finite It can be proved that $\mathcal{L} = L(M_{\mathcal{L}})$.

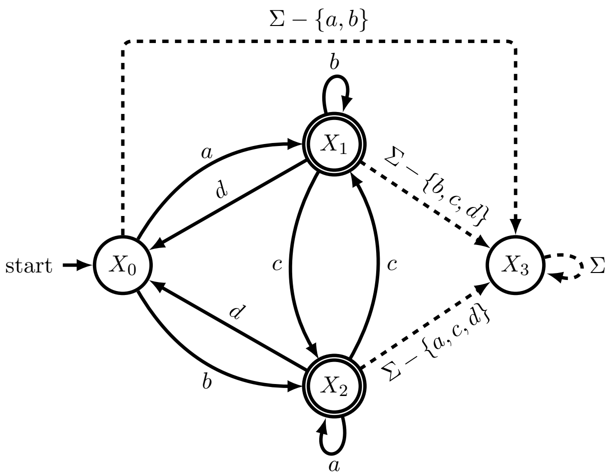


Figure 3: The relationship between automata and finite partition

end