tphols-2011

By xingyuan

February 9, 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 $while\text{-}option :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$ **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 = {s₁ @ s₂ | s₁ s₂. s₁ ∈ A ∧ s₂ ∈ 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 ∪ B) ;; C = (A ;; C) ∪ (B ;; C)
unfolding Seq-def by auto

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

lemma seq-intro:
  assumes a: x ∈ A y ∈ B
  shows x @ y ∈ 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
$A \uparrow 0 = \{[]\}$
| $A \uparrow (Suc\ n) = A ;; (A \uparrow n)$

definition
Star :: *lang* ⇒ *lang* (-⋆ [*101*] *102*)
where
$A\star \equiv (\bigcup n.\ A \uparrow n)$

lemma *star-start*[*intro*]:
shows $[] \in A\star$
proof −
have $[] \in A \uparrow 0$ **by** *auto*
then show $[] \in A\star$ **unfolding** *Star-def* **by** *blast*
qed

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

lemma *star-induct*[*consumes 1*, *case-names start step*]:
assumes *a*: $x \in A\star$
and *b*: $P\ []$
and *c*: $\bigwedge s1\ s2.\ [\![s1 \in A;\ s2 \in A\star;\ P\ s2]\!] \Longrightarrow P\ (s1\ @\ s2)$
shows $P\ x$
proof −
from *a* **obtain** *n* **where** $x \in A \uparrow n$ **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: (⋃ n. A ↑ n) ;; B = (⋃ n. (A ↑ 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 ∪ 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 arden:
  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 arden-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 $s \in X$
then have $s \in X$;; $(A \uparrow Suc\ k) \cup (\bigcup m{\in}\{0..k\}.\ B$;; $(A \uparrow m))$
using *arden-helper*[*OF eq*, *of k*] **by** *auto*
then have $s \in (\bigcup m{\in}\{0..k\}.\ B$;; $(A \uparrow m))$ **using** *not-in* **by** *auto*
moreover
have $(\bigcup m{\in}\{0..k\}.\ B$;; $(A \uparrow m)) \subseteq (\bigcup n.\ B$;; $(A \uparrow n))$ **by** *auto*
ultimately
have $s \in B$;; $A\star$
unfolding *seq-Union-left Star-def* **by** *auto* **}**
then have $X \subseteq B$;; $A\star$ **by** *auto*
ultimately
show $X = B$;; $A\star$ **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:: $'a \Rightarrow lang$

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 $L\ (\biguplus rs) = \bigcup\ (L\ \text{`}\ rs)$
apply(*rule set-eqI*)
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

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 ≈*A*, the set *finals A* singles out those which contains the strings from *A*.

definition
finals :: *lang* ⇒ *lang set*
where
finals A ≡ {≈*A* `` {*x*} | *x* . *x* ∈ *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* ⊆ (*UNIV* // ≈*A*)
unfolding *finals-def*
unfolding *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 = $\bigcup$ (*L* ‘ *rhs*)
end

definition
 trns-of rhs X ≡ {*Trn X r* | *r*. *Trn X r* ∈ *rhs*}

Transitions between equivalence classes

definition
 transition :: *lang* ⇒ *rexp* ⇒ *lang* ⇒ *bool* (- ⊨-⇒- [*100*,*100*,*100*] *100*)
where
 Y ⊨*r*⇒ *X* ≡ *Y* ;; (*L r*) ⊆ *X*

Initial equational system

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

definition
 eqs 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
$attach\text{-}rexp :: rexp \Rightarrow rhs\text{-}item \Rightarrow rhs\text{-}item$
where
$attach\text{-}rexp\ r\ (Lam\ rexp) \ \ = Lam\ (SEQ\ rexp\ r)$
$|\ attach\text{-}rexp\ r\ (Trn\ X\ rexp) = Trn\ X\ (SEQ\ rexp\ r)$

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

definition
$arden\text{-}op\ X\ rhs \equiv$
$\quad append\text{-}rhs\text{-}rexp\ (rhs - trns\text{-}of\ rhs\ X)\ (STAR\ (\biguplus\ \{r.\ Trn\ X\ r \in rhs\}))$

9 Substitution Operation on equations

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

definition
$subst\text{-}op\ rhs\ X\ xrhs \equiv$
$\quad (rhs - (trns\text{-}of\ rhs\ X)) \cup (append\text{-}rhs\text{-}rexp\ xrhs\ (\biguplus\ \{r.\ Trn\ X\ r \in rhs\}))$

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

definition
$subst\text{-}op\text{-}all\ ES\ X\ xrhs \equiv \{(Y,\ subst\text{-}op\ yrhs\ X\ xrhs)\ |\ Y\ yrhs.\ (Y,\ yrhs) \in ES\}$

10 While-combinator

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
$remove\text{-}op\ ES\ Y\ yrhs \equiv$
$\quad subst\text{-}op\text{-}all\ \ (ES - \{(Y,\ yrhs)\})\ Y\ (arden\text{-}op\ Y\ yrhs)$

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

definition
$iter\ X\ ES \equiv (let\ (Y,\ yrhs) = SOME\ (Y,\ yrhs).\ (Y,\ yrhs) \in ES \wedge (X \neq Y)$
$\quad\quad in\ remove\text{-}op\ ES\ Y\ yrhs)$

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

definition
$reduce\ X\ ES \equiv while\ (\lambda\ ES.\ card\ ES \neq 1)\ (iter\ X)\ ES$

Since the *while* combinator from HOL library is used to implement *reduce X ES*, the induction principle *while-rule* is used to proved the desired properties of *reduce 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 onece in *ES*.

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

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

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

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

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

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

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

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

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

definition
$classes\text{-}of\ rhs \equiv \{X.\ \exists\ r.\ Trn\ X\ r \in rhs\}$

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

definition
$lefts\text{-}of\ ES \equiv \{Y \mid Y\ yrhs.\ (Y,\ yrhs) \in ES\}$

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

definition
invariant ES ≡ *valid-eqns ES* ∧ *finite ES* ∧ *distinct-equas ES* ∧ *ardenable ES* ∧

finite-rhs ES ∧ *self-contained ES*

11.1 The proof of this direction

11.1.1 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-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:rhs-nonempty rhs
```

shows [] ∉ *L* (⊎ {*r*. *Trn X r* ∈ *rhs*})
using *finite nonempty rhs-nonempty-def*
using *finite-Trn*[*OF finite*]
by (*auto*)

lemma [*intro*!]:
P (*Trn X r*) ⟹ (∃ *a*. (∃ *r*. *a* = *Trn X r* ∧ *P a*)) **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*, *auto*)
apply(*rule-tac x*= *Trn X xa* **in** *exI*)
apply(*auto simp*: *Seq-def*)
done
qed

lemma *rexp-of-lam-eq-lam-set*:
assumes *fin*: *finite rhs*
shows *L* (⊎{*r*. *Lam r* ∈ *rhs*}) = *L* ({*Lam r* | *r*. *Lam r* ∈ *rhs*})
proof −
have *finite* ({*r*. *Lam r* ∈ *rhs*}) **using** *fin* **by** (*rule finite-Lam*)
then show *?thesis* **by** *auto*
qed

lemma [*simp*]:
L (*attach-rexp r xb*) = *L xb* ;; *L r*
apply (*cases xb*, *auto simp*: *Seq-def*)
apply(*rule-tac* $x = s_1$ @ $s_1{}'$ **in** *exI*, *rule-tac* $x = s_2{}'$ **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 ∪ *classes-of B* = *classes-of* (*A* ∪ *B*)
by (*auto simp add*:*classes-of-def*)

lemma *lefts-of-union-distrib*:

lefts-of A ∪ *lefts-of B* = *lefts-of* (*A* ∪ *B*)
by (*auto simp*:*lefts-of-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*:
⟦*s* ∈ *X*; *X* ∈ *UNIV* // (≈*Lang*)⟧ ⟹ *X* = (≈*Lang*) `` {*s*}
by (*auto simp*:*quotient-def Image-def str-eq-rel-def*)

lemma *every-eqclass-has-transition*:
  assumes *has-str*: *s* @ [*c*] ∈ *X*
  and *in-CS*: *X* ∈ *UNIV* // (≈*Lang*)
  obtains *Y* **where** *Y* ∈ *UNIV* // (≈*Lang*) **and** *Y* ;; {[*c*]} ⊆ *X* **and** *s* ∈ *Y*
proof −
  def *Y* ≡ (≈*Lang*) `` {*s*}
  have *Y* ∈ *UNIV* // (≈*Lang*)
    unfolding *Y-def quotient-def* **by** *auto*
  moreover
  have *X* = (≈*Lang*) `` {*s* @ [*c*]}
    using *has-str in-CS defined-by-str* **by** *blast*
  then have *Y* ;; {[*c*]} ⊆ *X*
    unfolding *Y-def Image-def Seq-def*
    unfolding *str-eq-rel-def*
    by *clarsimp*
  moreover
  have *s* ∈ *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*) ∈ (*eqs* (*UNIV* // (≈*Lang*)))
  shows *X* = *L xrhs*
proof
  show *X* ⊆ *L xrhs*
  proof
    fix *x*
    assume (*1*): *x* ∈ *X*
    show *x* ∈ *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 $X \in UNIV$ // $(\approx Lang)$ **using** *X-in-eqs* **by** (*auto simp*:*eqs-def*)
then obtain Y
where $Y \in UNIV$ // $(\approx Lang)$
and Y ;; $\{[c]\} \subseteq X$
and *clist* $\in Y$
using *decom* (*1*) *every-eqclass-has-transition* **by** *blast*
hence
$x \in L$ $\{Trn\ Y\ (CHAR\ c)|\ Y\ c.\ Y \in UNIV$ // $(\approx Lang) \wedge Y \models (CHAR\ c) \Rightarrow X\}$
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* $\subseteq X$ **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* $\{Trn\ Y\ (CHAR\ c)\ |Y\ c.\ Y \in CS \wedge Y$;; $\{[c]\} \subseteq X\}$ (**is** *finite ?A*)
proof −
def $S \equiv \{(Y, c)|\ Y\ c.\ Y \in CS \wedge Y$;; $\{[c]\} \subseteq X\}$
def $h \equiv \lambda\ (Y, c).\ Trn\ Y\ (CHAR\ c)$
have *finite* (*CS* $\times$ (*UNIV*::*char set*)) **using** *finite* **by** *auto*
hence *finite S* **using** *S-def*
by (*rule-tac* $B = CS \times UNIV$ **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-invariant*:
assumes *finite-CS*: *finite* (*UNIV* // $(\approx Lang)$)
shows *invariant* (*eqs* (*UNIV* // $(\approx Lang)$))
proof −
have *finite* (*eqs* (*UNIV* // $(\approx Lang)$)) **using** *finite-CS*
by (*simp add*:*eqs-def*)
moreover have *distinct-equas* (*eqs* (*UNIV* // $(\approx Lang)$))
by (*simp add*:*distinct-equas-def eqs-def*)
moreover have *ardenable* (*eqs* (*UNIV* // $(\approx Lang)$))
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 *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*:*invariant-def*)
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-op-keeps-eq*:
  assumes *l-eq-r*: $X = L\ rhs$
  and *not-empty*: $[] \notin L\ (\biguplus\{r.\ Trn\ X\ r \in rhs\})$
  and *finite*: *finite rhs*
  shows $X = L\ (arden\text{-}op\ X\ rhs)$
proof −
  def $A \equiv L\ (\biguplus\{r.\ Trn\ X\ r \in rhs\})$
  def $b \equiv rhs - trns\text{-}of\ rhs\ X$
  def $B \equiv L\ b$
  have $X = B\ ;;\ A\star$
  proof−
    have $L\ rhs = L(trns\text{-}of\ rhs\ X \cup b)$ **by** (*auto simp*: *b-def trns-of-def*)
    also have $\ldots = X\ ;;\ A \cup B$
      unfolding *trns-of-def*
      unfolding *L-rhs-union-distrib*[*symmetric*]
      by (*simp only*: *lang-of-rexp-of finite B-def A-def*)
    finally show *?thesis*
      using *l-eq-r not-empty*
      apply(*rule-tac arden*[*THEN iffD1*])
      apply(*simp add*: *A-def*)
      apply(*simp*)
      done
  qed
  moreover have $L\ (arden\text{-}op\ X\ rhs) = (B\ ;;\ A\star)$
    by (*simp only*:*arden-op-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-op-keeps-finite*:
  finite rhs ⟹ *finite* (*arden-op X rhs*)

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

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

lemma *subst-op-keeps-eq*:
assumes *substor*: *X* = *L xrhs*
and *finite*: *finite rhs*
shows *L* (*subst-op rhs X xrhs*) = *L rhs* (**is** *?Left* = *?Right*)
proof−
def *A* $\equiv$ *L* (*rhs* $-$ *trns-of rhs X*)
have *?Left* = *A* $\cup$ *L* (*append-rhs-rexp xrhs* ($\biguplus$ {*r*. *Trn X r* $\in$ *rhs*}))
unfolding *subst-op-def*
unfolding *L-rhs-union-distrib*[*symmetric*]
by (*simp add*: *A-def*)
moreover have *?Right* = *A* $\cup$ *L* ({*Trn X r* | *r*. *Trn X r* $\in$ *rhs*})
proof−
have *rhs* = (*rhs* $-$ *trns-of rhs X*) $\cup$ (*trns-of rhs X*) **by** (*auto simp add*: *trns-of-def*)
thus *?thesis*
unfolding *A-def*
unfolding *L-rhs-union-distrib*
unfolding *trns-of-def*
by *simp*
qed
moreover have *L* (*append-rhs-rexp xrhs* ($\biguplus$ {*r*. *Trn X r* $\in$ *rhs*})) = *L* ({*Trn X r* | *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-op-keeps-finite-rhs*:
  ⟦*finite rhs*; *finite yrhs*⟧ ⟹ *finite* (*subst-op rhs Y yrhs*)
by (*auto simp*:*subst-op-def append-keeps-finite*)

lemma *subst-op-all-keeps-finite*:
  assumes *finite*:*finite* (*ES*:: (*string set* × *rhs-item set*) *set*)
  shows *finite* (*subst-op-all ES Y yrhs*)
proof −
  have *finite* {(*Ya*, *subst-op 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*, *subst-op 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*:*subst-op-all-def*)
qed

lemma *subst-op-all-keeps-finite-rhs*:
  ⟦*finite-rhs ES*; *finite yrhs*⟧ ⟹ *finite-rhs* (*subst-op-all ES Y yrhs*)
by (*auto intro*:*subst-op-keeps-finite-rhs simp add*:*subst-op-all-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-op-removes-cl*:
  classes-of (*arden-op Y yrhs*) = *classes-of yrhs* − {*Y*}
apply (*simp add*:*arden-op-def append-rhs-keeps-cls trns-of-def*)
by (*auto simp*:*classes-of-def*)

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

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

lemma *subst-op-all-keeps-self-contained*:
  fixes *Y*

```
  assumes sc: self-contained (ES ∪ {(Y, yrhs)}) (is self-contained ?A)
  shows self-contained (subst-op-all ES Y (arden-op Y yrhs))
                                           (is self-contained ?B)
proof−
  { fix X xrhs'
    assume (X, xrhs') ∈ ?B
    then obtain xrhs
      where xrhs-xrhs': xrhs' = subst-op xrhs Y (arden-op Y yrhs)
      and X-in: (X, xrhs) ∈ ES by (simp add:subst-op-all-def, blast)
    have classes-of xrhs' ⊆ lefts-of ?B
    proof−
     have lefts-of ?B = lefts-of ES by (auto simp add:lefts-of-def subst-op-all-def)
      moreover have classes-of xrhs' ⊆ lefts-of ES
      proof−
        have classes-of xrhs' ⊆
                    classes-of xrhs ∪ classes-of (arden-op Y yrhs) − {Y}
        proof−
          have Y ∉ classes-of (arden-op Y yrhs)
            using arden-op-removes-cl by simp
          thus ?thesis using xrhs-xrhs' by (auto simp:subst-op-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-op Y yrhs) ⊆ lefts-of ES ∪ {Y}
          using sc
          by (auto simp add:arden-op-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:subst-op-all-def self-contained-def)
qed

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

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

lemma subst-op-all-card-le:
  assumes finite: finite (ES::(string set × rhs-item set) set)
  shows card (subst-op-all ES Y yrhs) <= card ES
proof−
  def f ≡ λ x. ((fst x)::string set, subst-op (snd x) Y yrhs)
  have subst-op-all ES Y yrhs = f ‘ ES
    apply (auto simp:subst-op-all-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 *subst-op-all-cls-remains*:
$(X, xrhs) \in ES \Longrightarrow \exists\ xrhs'. (X, xrhs') \in (\textit{subst-op-all ES Y yrhs})$
by (*auto simp*:*subst-op-all-def*)

lemma *card-noteq-1-has-more*:
assumes *card*:*card* $S \neq 1$
and *e-in*: $e \in S$
and *finite*: *finite S*
obtains e' **where** $e' \in S \wedge e \neq e'$
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 $S - \{e\} \neq \{\}$ **using** *finite* **by** (*rule-tac notI*, *simp*)
thus $(\bigwedge e'.\ e' \in S \wedge e \neq e' \Longrightarrow thesis) \Longrightarrow thesis$ **by** *auto*
qed

lemma *iteration-step*:
assumes *Inv-ES*: *invariant ES*
and *X-in-ES*: $(X, xrhs) \in ES$
and *not-T*: *card* $ES \neq 1$
shows $(\textit{invariant (iter X ES)} \wedge (\exists\ xrhs'.(X, xrhs') \in (\textit{iter X ES})) \wedge$
$(\textit{iter X ES}, ES) \in \textit{measure card})$
proof −
have *finite-ES*: *finite ES* **using** *Inv-ES* **by** (*simp add*: *invariant-def*)
then obtain *Y yrhs*
where *Y-in-ES*: $(Y, yrhs) \in ES$ **and** *not-eq*: $(X, xrhs) \neq (Y, yrhs)$
using *not-T X-in-ES* **by** (*drule-tac card-noteq-1-has-more*, *auto*)
let *?ES'* = *iter X ES*
show *?thesis*
proof(*unfold iter-def remove-op-def*, *rule someI2* [**where** $a = (Y, yrhs)$], *clar-simp*)
from *X-in-ES Y-in-ES* **and** *not-eq* **and** *Inv-ES*
show $(Y, yrhs) \in ES \wedge X \neq Y$
by (*auto simp*: *invariant-def distinct-equas-def*)
next
fix *x*
let *?ES'* = *let* $(Y, yrhs) = x$ *in subst-op-all* $(ES - \{(Y, yrhs)\})$ *Y* (*arden-op Y yrhs*)
assume *prem*: *case x of* $(Y, yrhs) \Rightarrow (Y, yrhs) \in ES \wedge (X \neq Y)$
thus *invariant* (*?ES'*) $\wedge$ ($\exists$*xrhs'*. $(X, xrhs') \in$ *?ES'*) $\wedge$ (*?ES'*, *ES*) $\in$ *measure card*
proof(*cases x*, *simp*)

```
      fix Y yrhs
      assume h: (Y, yrhs) ∈ ES ∧  X ≠ Y
      show invariant (subst-op-all (ES − {(Y, yrhs)}) Y (arden-op Y yrhs)) ∧
            (∃ xrhs'. (X, xrhs') ∈ subst-op-all (ES − {(Y, yrhs)}) Y (arden-op Y
yrhs)) ∧
           card (subst-op-all (ES − {(Y, yrhs)}) Y (arden-op Y yrhs)) < card ES
      proof −
        have invariant (subst-op-all (ES − {(Y, yrhs)}) Y (arden-op Y yrhs))
        proof(rule subst-op-all-satisfy-invariant)
          from h have (Y, yrhs) ∈ ES by simp
          hence ES − {(Y, yrhs)} ∪ {(Y, yrhs)} = ES by auto
          with Inv-ES show invariant (ES − {(Y, yrhs)} ∪ {(Y, yrhs)}) by auto
        qed
        moreover have
           (∃ xrhs'. (X, xrhs') ∈ subst-op-all (ES − {(Y, yrhs)}) Y (arden-op Y
yrhs))
        proof(rule subst-op-all-cls-remains)
          from X-in-ES and h
          show (X, xrhs) ∈ ES − {(Y, yrhs)} by auto
        qed
        moreover have
          card (subst-op-all (ES − {(Y, yrhs)}) Y (arden-op Y yrhs)) < card ES
        proof(rule le-less-trans)
          show
            card (subst-op-all (ES − {(Y, yrhs)}) Y (arden-op Y yrhs)) ≤
                                                    card (ES − {(Y, yrhs)})
          proof(rule subst-op-all-card-le)
            show finite (ES − {(Y, yrhs)}) using finite-ES by auto
          qed
        next
          show card (ES − {(Y, yrhs)}) < card ES using finite-ES h
            by (auto simp:card-gt-0-iff intro:diff-Suc-less)
        qed
        ultimately show ?thesis
          by (auto dest: subst-op-all-card-le elim:le-less-trans)
      qed
    qed
  qed
qed
```

11.1.4 Conclusion of the proof

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

```
lemma reduce-x:
  assumes inv: invariant ES
  and contain-x: (X, xrhs) ∈ ES
  shows ∃ xrhs'. reduce X ES = {(X, xrhs')} ∧ invariant(reduce X ES)
proof −
```

```
  let ?Inv = λ ES. (invariant ES ∧ (∃ xrhs. (X, xrhs) ∈ ES))
  show ?thesis
  proof (unfold reduce-def,
       rule while-rule [where P = ?Inv and r = measure card])
    from inv and contain-x show ?Inv ES by auto
  next
    show wf (measure card) by simp
  next
    fix ES
    assume inv: ?Inv ES and crd: card ES ≠ 1
    show (iter X ES, ES) ∈ measure card
    proof −
      from inv obtain xrhs where x-in: (X, xrhs) ∈ ES by auto
      from inv have invariant ES by simp
      from iteration-step [OF this x-in crd]
      show ?thesis by auto
    qed
  next
    fix ES
    assume inv: ?Inv ES and crd: card ES ≠ 1
    thus ?Inv (iter X ES)
    proof −
      from inv obtain xrhs where x-in: (X, xrhs) ∈ ES by auto
      from inv have invariant ES by simp
      from iteration-step [OF this x-in crd]
      show ?thesis by auto
    qed
  next
    fix ES
    assume ?Inv ES and ¬ card ES ≠ 1
    thus ∃xrhs'. ES = {(X, xrhs')} ∧ invariant ES
      apply (auto, rule-tac x = xrhs in exI)
      by (auto simp: invariant-def dest!:card-Suc-Diff1 simp:card-eq-0-iff)
  qed
qed

lemma last-cl-exists-rexp:
  assumes Inv-ES: invariant {(X, xrhs)}
  shows ∃ (r::rexp). L r = X (is ∃ r. ?P r)
proof−
  def A ≡ arden-op X xrhs
  have ?P (⨄{r. Lam r ∈ A})
  proof −
    have L (⨄{r. Lam r ∈ A}) = L ({Lam r | r. Lam r ∈  A})
    proof(rule rexp-of-lam-eq-lam-set)
      show finite A
        unfolding A-def
        using Inv-ES
        by (rule-tac arden-op-keeps-finite)
```

```
          (auto simp add: invariant-def finite-rhs-def)
    qed
    also have ... = L A
    proof−
      have {Lam r | r. Lam r ∈ A} = A
      proof−
        have classes-of A = {} using Inv-ES
          unfolding A-def
          by (simp add:arden-op-removes-cl
                    self-contained-def invariant-def lefts-of-def)
        thus ?thesis
          unfolding A-def
          by (auto simp only: classes-of-def, case-tac x, auto)
      qed
      thus ?thesis by simp
    qed
    also have ... = X
    unfolding A-def
    proof(rule arden-op-keeps-eq [THEN sym])
      show X = L xrhs using Inv-ES
        by (auto simp only: invariant-def valid-eqns-def)
    next
      from Inv-ES show [] ∉ L (⊎{r. Trn X r ∈ xrhs})
        by(simp add: invariant-def ardenable-def rexp-of-empty finite-rhs-def)
    next
      from Inv-ES show finite xrhs
        by (simp add: invariant-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 −
  let ?ES = eqs (UNIV // ≈Lang)
  from X-in-CS
  obtain xrhs where (X, xrhs) ∈ ?ES
    by (auto simp:eqs-def init-rhs-def)
  from reduce-x [OF init-ES-satisfy-invariant [OF finite-CS] this]
  have ∃xrhs'. reduce X ?ES = {(X, xrhs')} ∧ invariant (reduce X ?ES) .
  then obtain xrhs' where invariant {(X, xrhs')} by auto
  from last-cl-exists-rexp [OF this]
  show ?thesis .
qed
```

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* (⨄ *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

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* ≤ *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*

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 $xs = ys @ [y] \vee xs \leq ys$
then show $xs \leq ys @ [y]$
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 ≤ ys ⟹ set xs ⊆ 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*: $xs \le ys$
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** $ys \le xs$ @ $[x]$ **by** (*simp add*: *strict-prefix-def*)
with *snoc* **have** *False* **by** *blast*
then show *?thesis* **..**
next
assume $xs \parallel ys$
with *snoc* **obtain** *as b bs c cs* **where** *neq*: $(b::'a) \neq c$
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*: $a \parallel b \Longrightarrow a$ @ $c \parallel b$ @ 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*

12.4 Postfix order on lists

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

$[\![c \le a - b;\ b \le a]\!] \Longrightarrow b\ @\ c \le a$
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 *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. 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*) $\Longrightarrow$ *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*:

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

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
```

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*

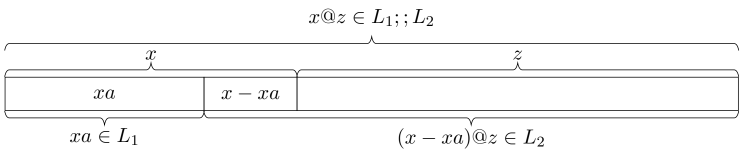
definition
  tag-str-ALT :: *lang* ⇒ *lang* ⇒ *string* ⇒ (*lang* × *lang*)
where
  tag-str-ALT L1 L2 = (λ*x*. (≈*L1* ‘‘ {*x*}, ≈*L2* ‘‘ {*x*}))

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

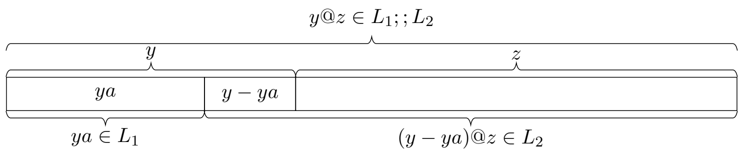
14.2.5 The inductive case for *SEQ*

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.

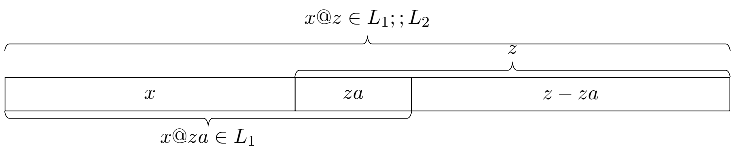
definition
  tag-str-SEQ :: *lang* ⇒ *lang* ⇒ *string* ⇒ (*lang* × *lang set*)



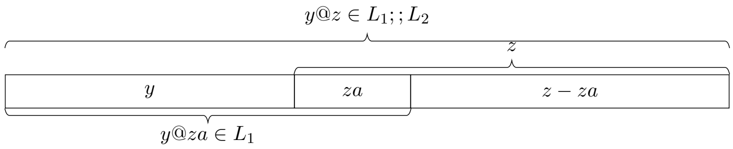
(a) First possible way to split $x@z$



(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* $\Longrightarrow \exists\ ya \leq y.\ (x\ @\ ya) \in L_1 \wedge (y - ya) \in L_2$
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 $\{\approx L_2\ ``\ \{x - xa\}\ |xa.\ xa \leq x \wedge xa \in L_1\} =$
$\{\approx L_2\ ``\ \{y - ya\}\ |ya.\ ya \leq y \wedge ya \in L_1\}$ (**is** *?Left* = *?Right*)
using *tag-xy* **unfolding** *tag-str-SEQ-def* **by** *simp*
— Since $xa \leq x$ 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: ≈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-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:

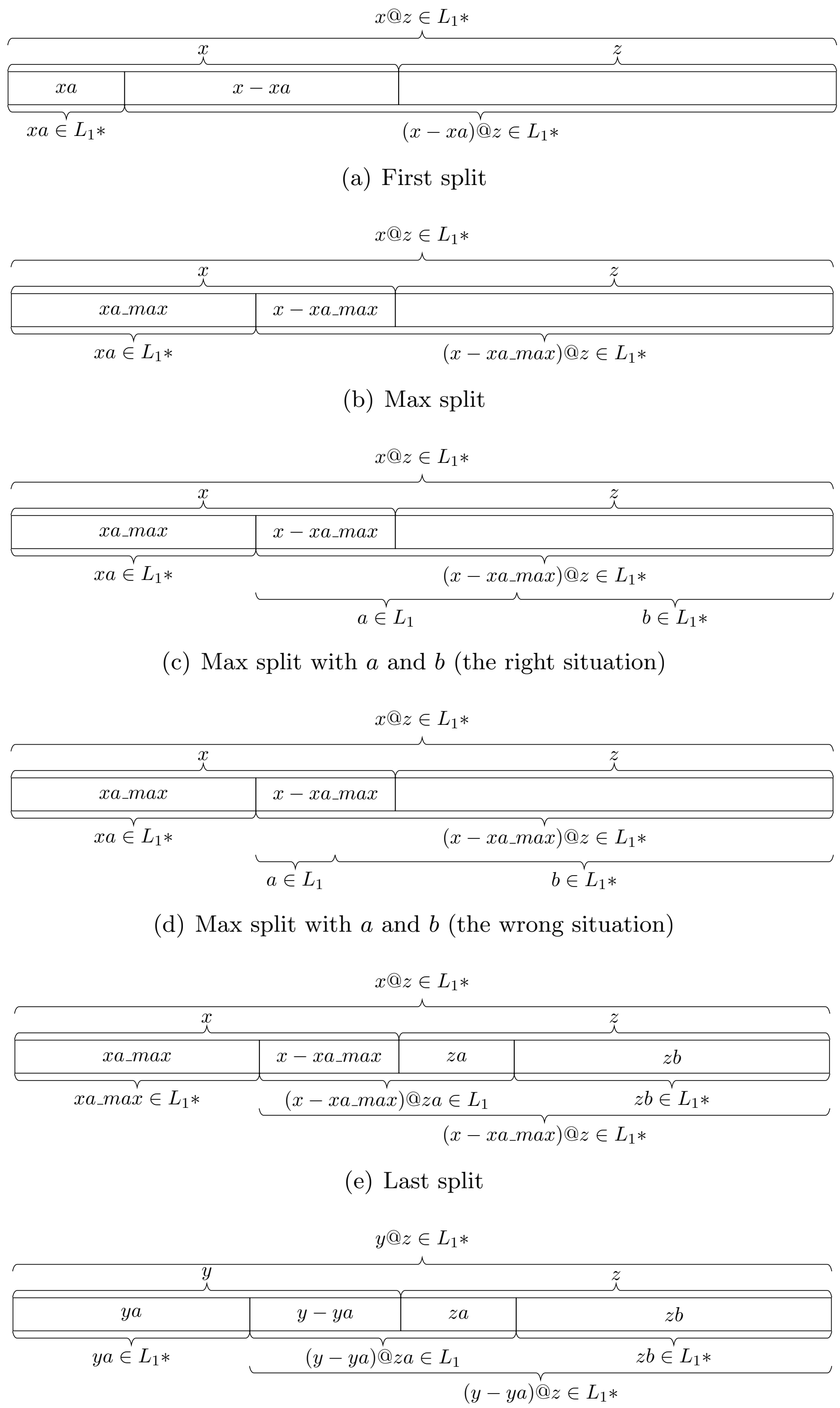
   (a) A y-prefix ya corresponding to xa can be found, which satisfies conditions: $ya \in L_1*$ and $(y - ya)@za \in L_1$.

   (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 = (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*:
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−
— As explained before, a pattern for just one direction needs to be dealt with:
{ **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 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*: $(x -$ *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*: $(x -$ *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 $\approx L_1$ ‘‘ $\{x -$ *xa-max*$\} \in$ *?left* **using** *h1 h2* **by** *auto*
ultimately have $\approx L_1$ ‘‘ $\{x -$ *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** $(x - xa\text{-}max)$ @ $z \neq []$
    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*: $tag\text{-}str\text{-}STAR\ L_1\ v = tag\text{-}str\text{-}STAR\ L_1\ w$
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*: $tag\text{-}str\text{-}STAR\ L_1\ x = tag\text{-}str\text{-}STAR\ L_1\ y$
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*: $x\text{-}max < x$
and *h2*: $x\text{-}max \in L_1\star$
and *h3*: $(x - x\text{-}max)\ @\ z \in L_1\star$
and *h4*: $\forall\ xa < x.\ xa \in L_1\star \wedge (x - xa)\ @\ z \in L_1\star \longrightarrow length\ xa \leq length\ x\text{-}max$
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 $?S \neq \{\}$ **using** *False xz-in-star*
by (*simp*, *rule-tac* $x = []$ **in** *exI*, *auto simp*:*strict-prefix-def*)
ultimately have $\exists\ max \in ?S.\ \forall\ a \in ?S.\ length\ a \leq length\ max$
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* // $\approx$*L1*)
 shows *finite* (*UNIV* // $\approx$(*L1*$\star$))
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* $\Longrightarrow$ *x* $\approx$(*L1*$\star$) *y*
  by (*rule tag-str-STAR-injI*)
next
 have $*$: *finite* (*Pow* (*UNIV* // $\approx$*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 *rexp-imp-finite*:
 fixes *r*::*rexp*
 shows *finite* (*UNIV* // $\approx$(*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 $\Rightarrow$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 set aside an unknown for every DFA state and describe the state-trasition relationship by charateristic equations. By solving the equational system such obtained, regular expressions characterizing DFA states are obtained. There are choices of how DFA states can be characterized. The first is to characterize a DFA state by the set of striings leading from the state in question into accepting states. The other choice is to characterize a DFA state by the set of strings leading from initial state into the state in question. For the first choice, the lauguage recognized by a DFA can be characterized by the regular expression characterizing initial state, while in the second choice, the languaged of the DFA can be characterized by the summation of regular expressions of all accepting states.

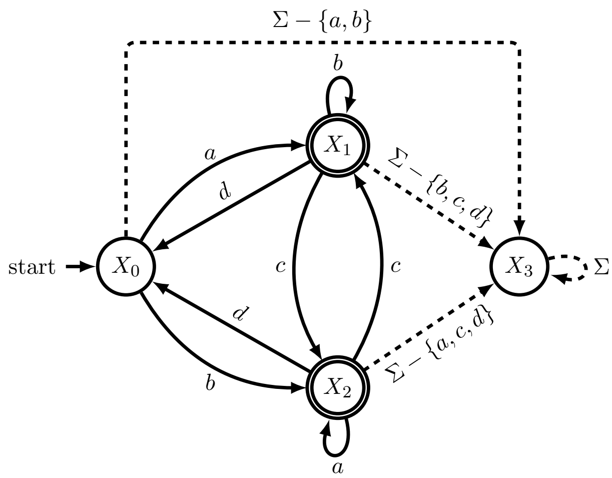


Figure 3: The relationship between automata and finite partition

end