tphols-2011

By xingyuan

February 20, 2012

Contents

1 Regular sets

theory *Regular-Set*
imports *Main*
begin

type-synonym *'a lang* = *'a list set*

definition *conc* :: *'a lang* ⇒ *'a lang* ⇒ *'a lang* (**infixr** @@ *75*) **where**
A @@ *B* = {*xs*@*ys* | *xs ys*. *xs*:*A* & *ys*:*B*}

overloading *lang-pow* == *compow* :: *nat* ⇒ *'a lang* ⇒ *'a lang*
begin
  primrec *lang-pow* :: *nat* ⇒ *'a lang* ⇒ *'a lang* **where**
  lang-pow 0 A = {[]} |
  lang-pow (*Suc n*) *A* = *A* @@ (*lang-pow n A*)
end

definition *star* :: *'a lang* ⇒ *'a lang* **where**
star A = (⋃*n*. *A* ^^ *n*)

1.1 *op* @@

lemma *concI*[*simp*,*intro*]: *u* : *A* ⟹ *v* : *B* ⟹ *u*@*v* : *A* @@ *B*
by (*auto simp add*: *conc-def*)

lemma *concE*[*elim*]:

assumes $w \in A$ @@ B
obtains u v **where** $u \in A$ $v \in B$ $w = u@v$
using *assms* **by** (*auto simp*: *conc-def*)

lemma *conc-mono*: $A \subseteq C \Longrightarrow B \subseteq D \Longrightarrow A$ @@ $B \subseteq C$ @@ D
by (*auto simp*: *conc-def*)

lemma *conc-empty*[*simp*]: **shows** $\{\}$ @@ $A = \{\}$ **and** A @@ $\{\} = \{\}$
by *auto*

lemma *conc-epsilon*[*simp*]: **shows** $\{[]\}$ @@ $A = A$ **and** A @@ $\{[]\} = A$
by (*simp-all add*:*conc-def*)

lemma *conc-assoc*: (A @@ B) @@ $C = A$ @@ (B @@ C)
by (*auto elim*!: *concE*) (*simp only*: *append-assoc*[*symmetric*] *concI*)

lemma *conc-Un-distrib*:
shows A @@ $(B \cup C) = A$ @@ $B \cup A$ @@ C
and $(A \cup B)$ @@ $C = A$ @@ $C \cup B$ @@ C
by *auto*

lemma *conc-UNION-distrib*:
shows A @@ *UNION I M* = *UNION I* (%i. A @@ M i)
and *UNION I M* @@ A = *UNION I* (%i. M i @@ A)
by *auto*

1.2 A^n

lemma *lang-pow-add*: A ^^ $(n + m) = A$ ^^ n @@ A ^^ m
by (*induct n*) (*auto simp*: *conc-assoc*)

lemma *lang-pow-empty*: $\{\}$ ^^ n = (*if* $n = 0$ *then* $\{[]\}$ *else* $\{\}$)
by (*induct n*) *auto*

lemma *lang-pow-empty-Suc*[*simp*]: ($\{\}$::'*a lang*) ^^ *Suc* $n = \{\}$
by (*simp add*: *lang-pow-empty*)

lemma *conc-pow-comm*:
shows A @@ (A ^^ n) = (A ^^ n) @@ A
by (*induct n*) (*simp-all add*: *conc-assoc*[*symmetric*])

lemma *length-lang-pow-ub*:
ALL w : A. *length* $w \leq k \Longrightarrow w$: A^^$n \Longrightarrow$ *length* $w \leq k*n$
by(*induct n arbitrary*: *w*) (*fastsimp simp*: *conc-def*)+

lemma *length-lang-pow-lb*:
ALL w : A. *length* $w \geq k \Longrightarrow w$: A^^$n \Longrightarrow$ *length* $w \geq k*n$
by(*induct n arbitrary*: *w*) (*fastsimp simp*: *conc-def*)+

1.3 *star*

```
lemma star-if-lang-pow[simp]: w : A ^^ n ⟹ w : star A
by (auto simp: star-def)

lemma Nil-in-star[iff]: [] : star A
proof (rule star-if-lang-pow)
  show [] : A ^^ 0 by simp
qed

lemma star-if-lang[simp]: assumes w : A shows w : star A
proof (rule star-if-lang-pow)
  show w : A ^^ 1 using ‹w : A› by simp
qed

lemma append-in-starI[simp]:
assumes u : star A and v : star A shows u@v : star A
proof –
  from ‹u : star A› obtain m where u : A ^^ m by (auto simp: star-def)
  moreover
  from ‹v : star A› obtain n where v : A ^^ n by (auto simp: star-def)
  ultimately have u@v : A ^^ (m+n) by (simp add: lang-pow-add)
  thus ?thesis by simp
qed

lemma conc-star-star: star A @@ star A = star A
by (auto simp: conc-def)

lemma conc-star-comm:
  shows A @@ star A = star A @@ A
unfolding star-def conc-pow-comm conc-UNION-distrib
by simp

lemma star-induct[consumes 1, case-names Nil append, induct set: star]:
assumes w : star A
  and P []
  and step: !!u v. u : A ⟹ v : star A ⟹ P v ⟹ P (u@v)
shows P w
proof –
  { fix n have w : A ^^ n ⟹ P w
    by (induct n arbitrary: w) (auto intro: ‹P []› step star-if-lang-pow) }
  with ‹w : star A› show P w by (auto simp: star-def)
qed

lemma star-empty[simp]: star {} = {[]}
by (auto elim: star-induct)

lemma star-epsilon[simp]: star {[]} = {[]}
by (auto elim: star-induct)
```

```
lemma star-idemp[simp]: star (star A) = star A
by (auto elim: star-induct)

lemma star-unfold-left: star A = A @@ star A ∪ {[]} (is ?L = ?R)
proof
  show ?L ⊆ ?R by (rule, erule star-induct) auto
qed auto

lemma concat-in-star: set ws ⊆ A ⟹ concat ws : star A
by (induct ws) simp-all

lemma in-star-iff-concat:
  w : star A = (EX ws. set ws ⊆ A & w = concat ws)
  (is - = (EX ws. ?R w ws))
proof
  assume w : star A thus EX ws. ?R w ws
  proof induct
    case Nil have ?R [] [] by simp
    thus ?case ..
  next
    case (append u v)
    moreover
    then obtain ws where set ws ⊆ A ∧ v = concat ws by blast
    ultimately have ?R (u@v) (u#ws) by auto
    thus ?case ..
  qed
next
  assume EX us. ?R w us thus w : star A
  by (auto simp: concat-in-star)
qed

lemma star-conv-concat: star A = {concat ws|ws. set ws ⊆ A}
by (fastsimp simp: in-star-iff-concat)

lemma star-insert-eps[simp]: star (insert [] A) = star(A)
proof−
  { fix us
    have set us ⊆ insert [] A ⟹ EX vs. concat us = concat vs ∧ set vs ⊆ A
      (is ?P ⟹ EX vs. ?Q vs)
    proof
      let ?vs = filter (%u. u ≠ []) us
      show ?P ⟹ ?Q ?vs by (induct us) auto
    qed
  } thus ?thesis by (auto simp: star-conv-concat)
qed

lemma star-decom:
  assumes a: x ∈ star A x ≠ []
  shows ∃ a b. x = a @ b ∧ a ≠ [] ∧ a ∈ A ∧ b ∈ star A
```

```
using a by (induct rule: star-induct) (blast)+
```

1.4 Arden's Lemma

```
lemma arden-helper:
  assumes eq: X = A @@ X ∪ B
  shows X = (A ^^ Suc n) @@ X ∪ (⋃m≤n. (A ^^ m) @@ B)
proof (induct n)
  case 0
  show X = (A ^^ Suc 0) @@ X ∪ (⋃m≤0. (A ^^ m) @@ B)
    using eq by simp
next
  case (Suc n)
  have ih: X = (A ^^ Suc n) @@ X ∪ (⋃m≤n. (A ^^ m) @@ B) by fact
  also have ... = (A ^^ Suc n) @@ (A @@ X ∪ B) ∪ (⋃m≤n. (A ^^ m) @@
B) using eq by simp
  also have ... = (A ^^ Suc (Suc n)) @@ X ∪ ((A ^^ Suc n) @@ B) ∪ (⋃m≤n.
(A ^^ m) @@ B)
    by (simp add: conc-Un-distrib conc-assoc[symmetric] conc-pow-comm)
  also have ... = (A ^^ Suc (Suc n)) @@ X ∪ (⋃m≤Suc n. (A ^^ m) @@ B)
    by (auto simp add: le-Suc-eq)
  finally show X = (A ^^ Suc (Suc n)) @@ X ∪ (⋃m≤Suc n. (A ^^ m) @@ B)
.
qed

lemma Arden:
  assumes [] ∉ A
  shows X = A @@ X ∪ B ⟷ X = star A @@ B
proof
  assume eq: X = A @@ X ∪ B
  { fix w assume w : X
    let ?n = size w
    from ‹[] ∉ A› have ALL u : A. length u ≥ 1
      by (metis Suc-eq-plus1 add-leD2 le-0-eq length-0-conv not-less-eq-eq)
    hence ALL u : A^^(?n+1). length u ≥ ?n+1
      by (metis length-lang-pow-lb nat-mult-1)
    hence ALL u : A^^(?n+1)@@X. length u ≥ ?n+1
      by(auto simp only: conc-def length-append)
    hence w ∉ A^^(?n+1)@@X by auto
     hence w : star A @@ B using ‹w : X› using arden-helper[OF eq, where
n=?n]
      by (auto simp add: star-def conc-UNION-distrib)
  } moreover
  { fix w assume w : star A @@ B
    hence EX n. w : A^^n @@ B by(auto simp: conc-def star-def)
    hence w : X using arden-helper[OF eq] by blast
  } ultimately show X = star A @@ B by blast
next
  assume eq: X = star A @@ B
```

```
  have star A = A @@ star A ∪ {[]}
    by (rule star-unfold-left)
  then have star A @@ B = (A @@ star A ∪ {[]}) @@ B
    by metis
  also have ... = (A @@ star A) @@ B ∪ B
    unfolding conc-Un-distrib by simp
  also have ... = A @@ (star A @@ B) ∪ B
    by (simp only: conc-assoc)
  finally show X = A @@ X ∪ B
    using eq by blast
qed


lemma reversed-arden-helper:
  assumes eq: X = X @@ A ∪ B
  shows X = X @@ (A ^^ Suc n) ∪ (⋃m≤n. B @@ (A ^^ m))
proof (induct n)
  case 0
  show X = X @@ (A ^^ Suc 0) ∪ (⋃m≤0. B @@ (A ^^ m))
    using eq by simp
next
  case (Suc n)
  have ih: X = X @@ (A ^^ Suc n) ∪ (⋃m≤n. B @@ (A ^^ m)) by fact
  also have ... = (X @@ A ∪ B) @@ (A ^^ Suc n) ∪ (⋃m≤n. B @@ (A ^^ m))
using eq by simp
  also have ... = X @@ (A ^^ Suc (Suc n)) ∪ (B @@ (A ^^ Suc n)) ∪ (⋃m≤n.
B @@ (A ^^ m))
    by (simp add: conc-Un-distrib conc-assoc)
  also have ... = X @@ (A ^^ Suc (Suc n)) ∪ (⋃m≤Suc n. B @@ (A ^^ m))
    by (auto simp add: le-Suc-eq)
  finally show X = X @@ (A ^^ Suc (Suc n)) ∪ (⋃m≤Suc n. B @@ (A ^^ m))
.
qed

theorem reversed-Arden:
  assumes nemp: [] ∉ A
  shows X = X @@ A ∪ B ⟷ X = B @@ star A
proof
 assume eq: X = X @@ A ∪ B
  { fix w assume w : X
    let ?n = size w
    from ‹[] ∉ A› have ALL u : A. length u ≥ 1
      by (metis Suc-eq-plus1 add-leD2 le-0-eq length-0-conv not-less-eq-eq)
    hence ALL u : A^^(?n+1). length u ≥ ?n+1
      by (metis length-lang-pow-lb nat-mult-1)
    hence ALL u : X @@ A^^(?n+1). length u ≥ ?n+1
      by(auto simp only: conc-def length-append)
    hence w ∉ X @@ A^^(?n+1) by auto
     hence w : B @@ star A using ‹w : X› using reversed-arden-helper[OF eq,
```

```
where n=?n]
    by (auto simp add: star-def conc-UNION-distrib)
 } moreover
 { fix w assume w : B @@ star A
   hence EX n. w : B @@ A^^n by (auto simp: conc-def star-def)
   hence w : X using reversed-arden-helper[OF eq] by blast
 } ultimately show X = B @@ star A by blast
next
 assume eq: X = B @@ star A
 have star A = {[]} ∪ star A @@ A
   unfolding conc-star-comm[symmetric]
   by(metis Un-commute star-unfold-left)
 then have B @@ star A = B @@ ({[]} ∪ star A @@ A)
   by metis
 also have ... = B ∪ B @@ (star A @@ A)
   unfolding conc-Un-distrib by simp
 also have ... = B ∪ (B @@ star A) @@ A
   by (simp only: conc-assoc)
 finally show X = X @@ A ∪ B
   using eq by blast
qed


end
```

2 Regular expressions

```
theory Regular-Exp
imports Regular-Set
begin

datatype 'a rexp =
 Zero |
 One |
 Atom 'a |
 Plus ('a rexp) ('a rexp) |
 Times ('a rexp) ('a rexp) |
 Star ('a rexp)

primrec lang :: 'a rexp => 'a lang where
lang Zero = {} |
lang One = {[]} |
lang (Atom a) = {[a]} |
lang (Plus r s) = (lang r) Un (lang s) |
lang (Times r s) = conc (lang r) (lang s) |
lang (Star r) = star(lang r)

primrec atoms :: 'a rexp ⇒ 'a set
```

where
atoms Zero = {} |
atoms One = {} |
atoms (*Atom a*) = {*a*} |
atoms (*Plus r s*) = *atoms r* ∪ *atoms s* |
atoms (*Times r s*) = *atoms r* ∪ *atoms s* |
atoms (*Star r*) = *atoms r*

end

theory *Folds*
imports *Regular-Exp*
begin

3 "Summation" for regular expressions

To obtain equational system out of finite set of equivalence classes, a fold operation on finite sets *folds* is defined. The use of *SOME* makes *folds* more robust than the *fold* in the Isabelle library. The expression *folds f* makes sense when *f* is not *associative* and *commutitive*, while *fold f* does not.

definition
  folds :: (′*a* ⇒ ′*b* ⇒ ′*b*) ⇒ ′*b* ⇒ ′*a set* ⇒ ′*b*
where
  folds f z S ≡ *SOME x. fold-graph f z S x*

Plus-combination for a set of regular expressions

abbreviation
  Setalt :: ′*a rexp set* ⇒ ′*a rexp* (⊎- [*1000*] *999*)
where
  ⊎*A* ≡ *folds Plus Zero A*

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

lemma *folds-plus-simp* [*simp*]:
  fixes *rs*::(′*a rexp*) *set*
  assumes *a*: *finite rs*
  shows *lang* (⊎*rs*) = ⋃ (*lang* ‘ *rs*)
unfolding *folds-def*
apply(*rule set-eqI*)
apply(*rule someI2-ex*)
apply(*rule-tac finite-imp-fold-graph*[*OF a*])
apply(*erule fold-graph.induct*)
apply(*auto*)
done

end

4 A general “while” combinator

theory *While-Combinator*
imports *Main*
begin

4.1 Partial version

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

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

lemma while-option-stop2:
 while-option b c s = Some t ⟹ EX k. t = (c^^k) s ∧ ¬ b t
```

apply(*simp add*: *while-option-def split*: *if-splits*)
by (*metis* (*lifting*) *LeastI-ex*)

lemma *while-option-stop*: *while-option b c s* = *Some t* ⟹ ~ *b t*
by(*metis while-option-stop2*)

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

4.2 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. [| $P\ s$; $\neg\ b\ s$ |] ==> $Q\ s$;
    wf r;
    !!s. [| $P\ s$; $b\ s$ |] ==> $(c\ s,\ s) \in r$ |] ==>
  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

Proving termination:

theorem *wf-while-option-Some*:
  assumes *wf* $\{(t,\ s).\ (P\ s \wedge b\ s) \wedge t = c\ s\}$
  and !!s. $P\ s \Longrightarrow b\ s \Longrightarrow P(c\ s)$ **and** $P\ s$
  shows *EX* t. *while-option* $b\ c\ s$ = *Some* t
using *assms*(*1*,*3*)
apply (*induct* s)
using *assms*(*2*)
apply (*subst while-option-unfold*)
apply *simp*
done

theorem *measure-while-option-Some*: **fixes** f :: $'s \Rightarrow$ *nat*
shows (!!s. $P\ s \Longrightarrow b\ s \Longrightarrow P(c\ s) \wedge f(c\ s) < f\ s$)
  $\Longrightarrow P\ s \Longrightarrow$ *EX* t. *while-option* $b\ c\ s$ = *Some* t
by(*blast intro*: *wf-while-option-Some*[*OF wf-if-measure*, *of P b f*])

Kleene iteration starting from the empty set and assuming some finite bounding set:

lemma *while-option-finite-subset-Some*: **fixes** C :: $'a$ *set*
  assumes *mono* f **and** !!X. $X \subseteq C \Longrightarrow f\ X \subseteq C$ **and** *finite* C
  shows $\exists\,P$. *while-option* $(\lambda A.\ f\ A \neq A)$ f {} = *Some* P
proof(*rule measure-while-option-Some*[**where**
   f= %A::$'a$ *set*. *card* C − *card* A **and** P= %A. $A \subseteq C \wedge A \subseteq f\ A$ **and** s= {}])
  fix A **assume** A: $A \subseteq C \wedge A \subseteq f\ A\ f\ A \neq A$
  show $(f\ A \subseteq C \wedge f\ A \subseteq f\ (f\ A)) \wedge$ *card* C − *card* $(f\ A)$ < *card* C − *card* A
   (**is** *?L* ∧ *?R*)
  proof
   show *?L* **by**(*metis A*(*1*) *assms*(*2*) *monoD*[*OF* ‹*mono f*›])
  show *?R* **by** (*metis A assms*(*2*,*3*) *card-seteq diff-less-mono2 equalityI linorder-le-less-linear rev-finite-subset*)
  qed
qed *simp*

lemma *lfp-the-while-option*:
assumes *mono f* **and** !!X. $X \subseteq C \Longrightarrow f\ X \subseteq C$ **and** *finite C*
shows *lfp f* = *the*(*while-option* (λA. $f\ A \neq A$) f {})
proof−
obtain P **where** *while-option* (λA. $f\ A \neq A$) f {} = *Some P*
using *while-option-finite-subset-Some*[*OF assms*] **by** *blast*
with *while-option-stop2*[*OF this*] *lfp-Kleene-iter*[*OF assms*(*1*)]
show *?thesis* **by** *auto*
qed

end

theory *Myhill-1*
imports *Folds*
~~/*src*/*HOL*/*Library*/*While-Combinator*
begin

5 First direction of MN: *finite partition ⇒ regular language*

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

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 :: *'a lang* ⇒ (*'a list* × *'a list*) *set* (≈- [*100*] *100*)
where
$\approx A \equiv \{(x, y).\ (\forall z.\ x\ @\ z \in A \longleftrightarrow y\ @\ z \in A)\}$

abbreviation
str-eq-applied :: *'a list* ⇒ *'a lang* ⇒ *'a list* ⇒ *bool* (- ≈- -)
where
$x \approx A\ y \equiv (x, y) \in \approx A$

definition
finals :: *'a lang* ⇒ *'a lang set*
where
finals $A \equiv \{\approx A\ ``\ \{s\} \mid s\ .\ s \in A\}$

lemma *lang-is-union-of-finals*:
shows $A = \bigcup$ *finals A*
unfolding *finals-def*

unfolding *Image-def*
unfolding *str-eq-def*
by (*auto*) (*metis append-Nil2*)

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

5.1 Equational systems

The two kinds of terms in the rhs of equations.

datatype *'a trm* =
   Lam 'a rexp
 | *Trn 'a lang 'a rexp*

fun
  lang-trm::*'a trm* ⇒ *'a lang*
where
  lang-trm (*Lam r*) = *lang r*
| *lang-trm* (*Trn X r*) = $X \cdot$ *lang r*

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

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

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

Transitions between equivalence classes

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

Initial equational system

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

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

5.2 Arden Operation on equations

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

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

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

5.3 Substitution Operation on equations

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

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

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

5.4 While-combinator and invariants

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

lemma *IterI2*:
assumes $(Y,\ yrhs) \in ES$
and $X \neq Y$
and $\bigwedge Y\ yrhs.\ [\![(Y,\ yrhs) \in ES;\ X \neq Y]\!] \Longrightarrow Q\ (Remove\ ES\ Y\ yrhs)$
shows $Q\ (Iter\ X\ ES)$
unfolding *Iter-def* **using** *assms*

by (*rule-tac a*=(*Y*, *yrhs*) **in** *someI2*) (*auto*)

abbreviation
Cond ES ≡ *card ES* ≠ *1*

definition
Solve X ES ≡ *while Cond* (*Iter X*) *ES*

definition
distinctness ES ≡
∀ *X rhs rhs'*. (*X*, *rhs*) ∈ *ES* ∧ (*X*, *rhs'*) ∈ *ES* ⟶ *rhs* = *rhs'*

definition
soundness ES ≡ ∀ (*X*, *rhs*) ∈ *ES*. *X* = *lang-rhs rhs*

definition
ardenable rhs ≡ (∀ *Y r*. *Trn Y r* ∈ *rhs* ⟶ [] ∉ *lang r*)

definition
ardenable-all ES ≡ ∀ (*X*, *rhs*) ∈ *ES*. *ardenable rhs*

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

lemma *finite-rhs-def2*:
finite-rhs ES = (∀ *X rhs*. (*X*, *rhs*) ∈ *ES* ⟶ *finite rhs*)
unfolding *finite-rhs-def* **by** *auto*

definition
rhss rhs ≡ {*X* | *X r*. *Trn X r* ∈ *rhs*}

definition
lhss ES ≡ {*Y* | *Y yrhs*. (*Y*, *yrhs*) ∈ *ES*}

definition
validity ES ≡ ∀ (*X*, *rhs*) ∈ *ES*. *rhss rhs* ⊆ *lhss ES*

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

lemma *lhss-union-distrib*:
shows *lhss* (*A* ∪ *B*) = *lhss A* ∪ *lhss B*
by (*auto simp add*: *lhss-def*)

definition
invariant ES ≡ *finite ES*

```
              ∧ finite-rhs ES
              ∧ soundness ES
              ∧ distinctness ES
              ∧ ardenable-all ES
              ∧ validity ES


lemma invariantI:
  assumes soundness ES finite ES distinctness ES ardenable-all ES
          finite-rhs ES validity ES
  shows invariant ES
using assms by (simp add: invariant-def)


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 trm-soundness:
  assumes finite:finite rhs
  shows lang-rhs ({Trn X r| r. Trn X r ∈ rhs}) = X · (lang (⨄{r. Trn X r ∈
rhs}))
proof −
  have finite {r. Trn X r ∈ rhs}
    by (rule finite-Trn[OF finite])
  then show lang-rhs ({Trn X r| r. Trn X r ∈ rhs}) = X · (lang (⨄{r. Trn X r
```

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

lemma *lang-of-append-rexp*:
 lang-trm (*Append-rexp r trm*) = *lang-trm trm* · *lang r*
by (*induct rule*: *Append-rexp.induct*)
  (*auto simp add*: *conc-assoc*)

lemma *lang-of-append-rexp-rhs*:
 lang-rhs (*Append-rexp-rhs rhs r*) = *lang-rhs rhs* · *lang r*
unfolding *Append-rexp-rhs-def*
by (*auto simp add*: *conc-def lang-of-append-rexp*)

5.5 Intial Equational Systems

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

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

lemma *l-eq-r-in-eqs*:
 assumes *X-in-eqs*: $(X, rhs) \in Init\ (UNIV\ //\ \approx A)$
 shows $X = $ *lang-rhs rhs*
proof
 show $X \subseteq$ *lang-rhs rhs*

proof
fix *x*
assume *in-X*: $x \in X$
{ assume *empty*: $x = []$
then have $x \in$ *lang-rhs rhs* **using** *X-in-eqs in-X*
unfolding *Init-def Init-rhs-def*
by *auto*
}
moreover
{ assume *not-empty*: $x \neq []$
then obtain *s c* **where** *decom*: $x = s @ [c]$
using *rev-cases* **by** *blast*
have $X \in UNIV // \approx A$ **using** *X-in-eqs* **unfolding** *Init-def* **by** *auto*
then obtain *Y* **where** $Y \in UNIV // \approx A$ $Y \cdot \{[c]\} \subseteq X$ $s \in Y$
using *decom in-X every-eqclass-has-transition* **by** *metis*
then have $x \in$ *lang-rhs* $\{Trn\ Y\ (Atom\ c)|\ Y\ c.\ Y \in UNIV // \approx A \land Y \models c \Rightarrow X\}$
unfolding *transition-def*
using *decom* **by** (*force simp add*: *conc-def*)
then have $x \in$ *lang-rhs rhs* **using** *X-in-eqs in-X*
unfolding *Init-def Init-rhs-def* **by** *simp*
}
ultimately show $x \in$ *lang-rhs rhs* **by** *blast*
qed
next
show *lang-rhs rhs* $\subseteq X$ **using** *X-in-eqs*
unfolding *Init-def Init-rhs-def transition-def*
by *auto*
qed

lemma *finite-Init-rhs*:
fixes *CS*::(('*a*::*finite*) *lang*) *set*
assumes *finite*: *finite CS*
shows *finite* (*Init-rhs CS X*)
proof−
def $S \equiv \{(Y, c)|\ Y\ c{::}'a.\ Y \in CS \land Y \cdot \{[c]\} \subseteq X\}$
def $h \equiv \lambda\ (Y, c{::}'a).\ Trn\ Y\ (Atom\ c)$
have *finite* (*CS* × (*UNIV*::('*a*::*finite*) *set*)) **using** *finite* **by** *auto*
then have *finite S* **using** *S-def*
by (*rule-tac* $B = CS \times UNIV$ **in** *finite-subset*) (*auto*)
moreover have $\{Trn\ Y\ (Atom\ c)\ |Y\ c{::}'a.\ Y \in CS \land Y \cdot \{[c]\} \subseteq X\} = h\ `\ S$
unfolding *S-def h-def image-def* **by** *auto*
ultimately
have *finite* $\{Trn\ Y\ (Atom\ c)\ |Y\ c.\ Y \in CS \land Y \cdot \{[c]\} \subseteq X\}$ **by** *auto*
then show *finite* (*Init-rhs CS X*) **unfolding** *Init-rhs-def transition-def* **by** *simp*
qed

lemma *Init-ES-satisfies-invariant*:
  fixes *A*::((*'a*::*finite*) *lang*)
  assumes *finite-CS*: *finite* (*UNIV* // ≈*A*)
  shows *invariant* (*Init* (*UNIV* // ≈*A*))
proof (*rule invariantI*)
  show *soundness* (*Init* (*UNIV* // ≈*A*))
    unfolding *soundness-def*
    using *l-eq-r-in-eqs* **by** *auto*
  show *finite* (*Init* (*UNIV* // ≈*A*)) **using** *finite-CS*
    unfolding *Init-def* **by** *simp*
  show *distinctness* (*Init* (*UNIV* // ≈*A*))
    unfolding *distinctness-def Init-def* **by** *simp*
  show *ardenable-all* (*Init* (*UNIV* // ≈*A*))
    unfolding *ardenable-all-def Init-def Init-rhs-def ardenable-def*
   by *auto*
  show *finite-rhs* (*Init* (*UNIV* // ≈*A*))
    using *finite-Init-rhs*[*OF finite-CS*]
    unfolding *finite-rhs-def Init-def* **by** *auto*
  show *validity* (*Init* (*UNIV* // ≈*A*))
    unfolding *validity-def Init-def Init-rhs-def rhss-def lhss-def*
    by *auto*
qed

5.6 Interations

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

unfolding *Arden-def A-def B-def b-def*
by (*simp only*: *lang-of-append-rexp-rhs lang.simps*)
finally show *X* = *lang-rhs* (*Arden X rhs*) **by** *simp*
qed

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

lemma *Arden-preserves-finite*:
finite rhs $\Longrightarrow$ *finite* (*Arden X rhs*)
by (*auto simp*: *Arden-def Append-preserves-finite*)

lemma *Append-preserves-ardenable*:
ardenable rhs $\Longrightarrow$ *ardenable* (*Append-rexp-rhs rhs r*)
apply (*auto simp*: *ardenable-def Append-rexp-rhs-def*)
by (*case-tac x*, *auto simp*: *conc-def*)

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

lemma *ardenable-set-union*:
⟦*ardenable rhs*; *ardenable rhs'*⟧ $\Longrightarrow$ *ardenable* (*rhs* ∪ *rhs'*)
by (*auto simp*:*ardenable-def*)

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

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

lemma *Subst-preserves-soundness*:
assumes *substor*: *X* = *lang-rhs xrhs*
and *finite*: *finite rhs*
shows *lang-rhs* (*Subst rhs X xrhs*) = *lang-rhs rhs* (**is** *?Left* = *?Right*)
proof−
def *A* ≡ *lang-rhs* (*rhs* − {*Trn X r* | *r*. *Trn X r* ∈ *rhs*})
have *?Left* = *A* ∪ *lang-rhs* (*Append-rexp-rhs xrhs* (⨄{*r*. *Trn X r* ∈ *rhs*}))
unfolding *Subst-def*
unfolding *lang-rhs-union-distrib*[*symmetric*]
by (*simp add*: *A-def*)
moreover have *?Right* = *A* ∪ *lang-rhs* {*Trn X r* | *r*. *Trn X r* ∈ *rhs*}
proof−
have *rhs* = (*rhs* − {*Trn X r* | *r*. *Trn X r* ∈ *rhs*}) ∪ ({*Trn X r* | *r*. *Trn X r*
∈ *rhs*}) **by** *auto*

thus *?thesis*
unfolding *A-def*
unfolding *lang-rhs-union-distrib*
by *simp*
qed
moreover
have *lang-rhs* (*Append-rexp-rhs xrhs* ($\biguplus$ {*r*. *Trn X r* ∈ *rhs*})) = *lang-rhs* {*Trn X r* | *r*. *Trn X r* ∈ *rhs*}
using *finite substor* **by** (*simp only*: *lang-of-append-rexp-rhs trm-soundness*)
ultimately show *?thesis* **by** *simp*
qed

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

lemma *Subst-all-preserves-finite*:
assumes *finite*: *finite ES*
shows *finite* (*Subst-all ES Y yrhs*)
proof −
def *eqns* ≡ {(*X*::'*a lang*, *rhs*) |*X rhs*. (*X*, *rhs*) ∈ *ES*}
def *h* ≡ λ(*X*::'*a lang*, *rhs*). (*X*, *Subst rhs Y yrhs*)
have *finite* (*h* ‘ *eqns*) **using** *finite h-def eqns-def* **by** *auto*
moreover
have *Subst-all ES Y yrhs* = *h* ‘ *eqns* **unfolding** *h-def eqns-def Subst-all-def* **by** *auto*
ultimately
show *finite* (*Subst-all ES Y yrhs*) **by** *simp*
qed

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

lemma *append-rhs-preserves-cls*:
rhss (*Append-rexp-rhs rhs r*) = *rhss rhs*
apply (*auto simp*: *rhss-def Append-rexp-rhs-def*)
apply (*case-tac xa*, *auto simp*: *image-def*)
by (*rule-tac x* = *Times ra r* **in** *exI*, *rule-tac x* = *Trn x ra* **in** *bexI*, *simp*+)

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

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

lemma *Subst-updates-cls*:
  $X \notin$ *rhss xrhs* $\Longrightarrow$
    rhss (*Subst rhs X xrhs*) = *rhss rhs* $\cup$ *rhss xrhs* $-$ $\{X\}$
apply (*simp only*:*Subst-def append-rhs-preserves-cls rhss-union-distrib*)
by (*auto simp*: *rhss-def*)

lemma *Subst-all-preserves-validity*:
  assumes *sc*: *validity* (*ES* $\cup$ $\{(Y, yrhs)\}$)   (**is** *validity ?A*)
  shows *validity* (*Subst-all ES Y* (*Arden Y yrhs*)) (**is** *validity ?B*)
proof $-$
  { fix *X xrhs'*
    assume (*X*, *xrhs'*) $\in$ *?B*
    then obtain *xrhs*
      where *xrhs-xrhs'*: *xrhs'* = *Subst xrhs Y* (*Arden Y yrhs*)
      and *X-in*: (*X*, *xrhs*) $\in$ *ES* **by** (*simp add*:*Subst-all-def*, *blast*)
    have *rhss xrhs'* $\subseteq$ *lhss ?B*
    proof$-$
      have *lhss ?B* = *lhss ES* **by** (*auto simp add*:*lhss-def Subst-all-def*)
      moreover have *rhss xrhs'* $\subseteq$ *lhss ES*
      proof$-$
        have *rhss xrhs'* $\subseteq$ *rhss xrhs* $\cup$ *rhss* (*Arden Y yrhs*) $-$ $\{Y\}$
        proof $-$
          have $Y \notin$ *rhss* (*Arden Y yrhs*)
            using *Arden-removes-cl* **by** *auto*
          thus *?thesis* **using** *xrhs-xrhs'* **by** (*auto simp*: *Subst-updates-cls*)
        qed
        moreover have *rhss xrhs* $\subseteq$ *lhss ES* $\cup$ $\{Y\}$ **using** *X-in sc*
          apply (*simp only*:*validity-def lhss-union-distrib*)
          by (*drule-tac x* = (*X*, *xrhs*) **in** *bspec*, *auto simp*:*lhss-def*)
        moreover have *rhss* (*Arden Y yrhs*) $\subseteq$ *lhss ES* $\cup$ $\{Y\}$
          using *sc*
          by (*auto simp add*: *Arden-removes-cl validity-def lhss-def*)
        ultimately show *?thesis* **by** *auto*
      qed
      ultimately show *?thesis* **by** *simp*
    qed
  } thus *?thesis* **by** (*auto simp only*:*Subst-all-def validity-def*)
qed

lemma *Subst-all-satisfies-invariant*:
  assumes *invariant-ES*: *invariant* (*ES* $\cup$ $\{(Y, yrhs)\}$)
  shows *invariant* (*Subst-all ES Y* (*Arden Y yrhs*))
proof (*rule invariantI*)
  have *Y-eq-yrhs*: *Y* = *lang-rhs yrhs*
    using *invariant-ES* **by** (*simp only*:*invariant-def soundness-def*, *blast*)
  have *finite-yrhs*: *finite yrhs*
    using *invariant-ES* **by** (*auto simp*:*invariant-def finite-rhs-def*)
  have *ardenable-yrhs*: *ardenable yrhs*
    using *invariant-ES* **by** (*auto simp*:*invariant-def ardenable-all-def*)

```
  show soundness (Subst-all ES Y (Arden Y yrhs))
  proof -
    have Y = lang-rhs (Arden Y yrhs)
      using Y-eq-yrhs invariant-ES finite-yrhs
      using finite-Trn[OF finite-yrhs]
      apply(rule-tac Arden-preserves-soundness)
      apply(simp-all)
      unfolding invariant-def ardenable-all-def ardenable-def
      apply(auto)
      done
    thus ?thesis using invariant-ES
      unfolding invariant-def finite-rhs-def2 soundness-def Subst-all-def
      by (auto simp add: Subst-preserves-soundness simp del: lang-rhs.simps)
  qed
  show finite (Subst-all ES Y (Arden Y yrhs))
    using invariant-ES by (simp add:invariant-def Subst-all-preserves-finite)
  show distinctness (Subst-all ES Y (Arden Y yrhs))
    using invariant-ES
    unfolding distinctness-def Subst-all-def invariant-def by auto
  show ardenable-all (Subst-all ES Y (Arden Y yrhs))
  proof -
    { fix X rhs
      assume (X, rhs) ∈ ES
      hence ardenable rhs  using invariant-ES
        by (auto simp add:invariant-def ardenable-all-def)
      with ardenable-yrhs
      have ardenable (Subst rhs Y (Arden Y yrhs))
        by (simp add:ardenable-yrhs
               Subst-preserves-ardenable Arden-preserves-ardenable)
    } thus ?thesis by (auto simp add:ardenable-all-def Subst-all-def)
  qed
  show finite-rhs (Subst-all ES Y (Arden Y yrhs))
  proof-
    have finite-rhs ES using invariant-ES
      by (simp add:invariant-def finite-rhs-def)
    moreover have finite (Arden Y yrhs)
    proof -
      have finite yrhs using invariant-ES
        by (auto simp:invariant-def finite-rhs-def)
      thus ?thesis using Arden-preserves-finite by auto
    qed
    ultimately show ?thesis
      by (simp add:Subst-all-preserves-finite-rhs)
  qed
  show validity (Subst-all ES Y (Arden Y yrhs))
   using invariant-ES Subst-all-preserves-validity by (auto simp add: invariant-def)
qed

lemma Remove-in-card-measure:
```

assumes *finite*: *finite ES*
and *in-ES*: $(X, rhs) \in ES$
shows (*Remove ES X rhs*, *ES*) $\in$ *measure card*
proof −
def $f \equiv \lambda\ x.\ ((fst\ x)::'a\ lang,\ Subst\ (snd\ x)\ X\ (Arden\ X\ rhs))$
def $ES' \equiv ES - \{(X, rhs)\}$
have *Subst-all* ES' X (*Arden X rhs*) $= f$ ‘ ES'
apply (*auto simp*: *Subst-all-def f-def image-def*)
by (*rule-tac* $x = (Y, yrhs)$ **in** *bexI*, *simp*+)
then have *card* (*Subst-all* ES' X (*Arden X rhs*)) $\leq$ *card* ES'
unfolding ES'*-def* **using** *finite* **by** (*auto intro*: *card-image-le*)
also have ... $<$ *card ES* **unfolding** ES'*-def*
using *in-ES finite* **by** (*rule-tac card-Diff1-less*)
finally show (*Remove ES X rhs*, *ES*) $\in$ *measure card*
unfolding *Remove-def* ES'*-def* **by** *simp*
qed

lemma *Subst-all-cls-remains*:
$(X, xrhs) \in ES \Longrightarrow \exists\ xrhs'.\ (X, xrhs') \in (Subst\text{-}all\ ES\ Y\ yrhs)$
by (*auto simp*: *Subst-all-def*)

lemma *card-noteq-1-has-more*:
assumes *card*:*Cond ES*
and *e-in*: $(X, xrhs) \in ES$
and *finite*: *finite ES*
shows $\exists (Y, yrhs) \in ES.\ (X, xrhs) \neq (Y, yrhs)$
proof−
have *card ES* $>$ *1* **using** *card e-in finite*
by (*cases card ES*) (*auto*)
then have *card* $(ES - \{(X, xrhs)\}) > 0$
using *finite e-in* **by** *auto*
then have $(ES - \{(X, xrhs)\}) \neq \{\}$ **using** *finite* **by** (*rule-tac notI*, *simp*)
then show $\exists (Y, yrhs) \in ES.\ (X, xrhs) \neq (Y, yrhs)$
by *auto*
qed

lemma *iteration-step-measure*:
assumes *Inv-ES*: *invariant ES*
and *X-in-ES*: $(X, xrhs) \in ES$
and *Cnd*: *Cond ES*
shows (*Iter X ES*, *ES*) $\in$ *measure card*
proof −
have *fin*: *finite ES* **using** *Inv-ES* **unfolding** *invariant-def* **by** *simp*
then obtain *Y yrhs*
where *Y-in-ES*: $(Y, yrhs) \in ES$ **and** *not-eq*: $(X, xrhs) \neq (Y, yrhs)$
using *Cnd X-in-ES* **by** (*drule-tac card-noteq-1-has-more*) (*auto*)
then have $(Y, yrhs) \in ES\ \ X \neq Y$
using *X-in-ES Inv-ES* **unfolding** *invariant-def distinctness-def*

```
    by auto
  then show (Iter X ES, ES) ∈ measure card
  apply(rule IterI2)
  apply(rule Remove-in-card-measure)
  apply(simp-all add: fin)
  done
qed

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

lemma iteration-step-ex:
  assumes Inv-ES: invariant ES
  and     X-in-ES: (X, xrhs) ∈ ES
  and     Cnd: Cond ES
  shows ∃ xrhs'. (X, xrhs') ∈ (Iter X ES)
proof −
  have finite-ES: finite ES using Inv-ES by (simp add: invariant-def)
  then obtain Y yrhs
    where (Y, yrhs) ∈ ES (X, xrhs) ≠ (Y, yrhs)
    using Cnd X-in-ES by (drule-tac card-noteq-1-has-more) (auto)
  then have (Y, yrhs) ∈ ES  X ≠ Y
    using X-in-ES Inv-ES unfolding invariant-def distinctness-def
    by auto
  then show ∃ xrhs'. (X, xrhs') ∈ (Iter X ES)
  apply(rule IterI2)
  unfolding Remove-def
  apply(rule Subst-all-cls-remains)
```

using *X-in-ES*
apply(*auto*)
done
qed

5.7 The conclusion of the first direction

lemma *Solve*:
fixes *A*::('*a*::*finite*) *lang*
assumes *fin*: *finite* (*UNIV* // ≈*A*)
and *X-in*: *X* ∈ (*UNIV* // ≈*A*)
shows ∃ *rhs*. *Solve X* (*Init* (*UNIV* // ≈*A*)) = {(*X*, *rhs*)} ∧ *invariant* {(*X*, *rhs*)}
proof −
def *Inv* ≡ λ*ES*. *invariant ES* ∧ (∃ *rhs*. (*X*, *rhs*) ∈ *ES*)
have *Inv* (*Init* (*UNIV* // ≈*A*)) **unfolding** *Inv-def*
using *fin X-in* **by** (*simp add*: *Init-ES-satisfies-invariant*, *simp add*: *Init-def*)
moreover
{ fix *ES*
assume *inv*: *Inv ES* **and** *crd*: *Cond ES*
then have *Inv* (*Iter X ES*)
unfolding *Inv-def*
by (*auto simp add*: *iteration-step-invariant iteration-step-ex*) **}**
moreover
{ fix *ES*
assume *inv*: *Inv ES* **and** *not-crd*: ¬*Cond ES*
from *inv* **obtain** *rhs* **where** (*X*, *rhs*) ∈ *ES* **unfolding** *Inv-def* **by** *auto*
moreover
from *not-crd* **have** *card ES* = *1* **by** *simp*
ultimately
have *ES* = {(*X*, *rhs*)} **by** (*auto simp add*: *card-Suc-eq*)
then have ∃ *rhs'*. *ES* = {(*X*, *rhs'*)} ∧ *invariant* {(*X*, *rhs'*)} **using** *inv*
unfolding *Inv-def* **by** *auto* **}**
moreover
have *wf* (*measure card*) **by** *simp*
moreover
{ fix *ES*
assume *inv*: *Inv ES* **and** *crd*: *Cond ES*
then have (*Iter X ES*, *ES*) ∈ *measure card*
unfolding *Inv-def*
apply(*clarify*)
apply(*rule-tac iteration-step-measure*)
apply(*auto*)
done }
ultimately
show ∃ *rhs*. *Solve X* (*Init* (*UNIV* // ≈*A*)) = {(*X*, *rhs*)} ∧ *invariant* {(*X*, *rhs*)}

unfolding *Solve-def* **by** (*rule while-rule*)
qed

```
lemma every-eqcl-has-reg:
  fixes A::('a::finite) lang
  assumes finite-CS: finite (UNIV // ≈A)
  and X-in-CS: X ∈ (UNIV // ≈A)
  shows ∃ r. X = lang r
proof −
  from finite-CS X-in-CS
  obtain xrhs where Inv-ES: invariant {(X, xrhs)}
    using Solve by metis

  def A ≡ Arden X xrhs
  have rhss xrhs ⊆ {X} using Inv-ES
    unfolding validity-def invariant-def rhss-def lhss-def
    by auto
  then have rhss A = {} unfolding A-def
    by (simp add: Arden-removes-cl)
  then have eq: {Lam r | r. Lam r ∈ A} = A unfolding rhss-def
    by (auto, case-tac x, auto)

  have finite A using Inv-ES unfolding A-def invariant-def finite-rhs-def
    using Arden-preserves-finite by auto
  then have fin: finite {r. Lam r ∈ A} by (rule finite-Lam)

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

lemma bchoice-finite-set:
  assumes a: ∀ x ∈ S. ∃ y. x = f y
  and     b: finite S
  shows ∃ ys. (⋃ S) = ⋃(f ‘ ys) ∧ finite ys
using bchoice[OF a] b
apply(erule-tac exE)
apply(rule-tac x=fa ‘ S in exI)
apply(auto)
done

theorem Myhill-Nerode1:
  fixes A::('a::finite) lang
  assumes finite-CS: finite (UNIV // ≈A)
  shows   ∃ r. A = lang r
proof −
  have fin: finite (finals A)
```

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

end

6 List prefixes and postfixes

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

6.1 Prefix order on lists

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

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

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

definition
$bot = []$

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

end

lemma *prefixI* [*intro?*]: $ys = xs\ @\ zs \Longrightarrow xs \leq ys$
unfolding *prefix-def* **by** *blast*

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

lemma *strict-prefixI'* [*intro?*]: $ys = xs\ @\ z\ \#\ zs \Longrightarrow xs < ys$

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*

6.2 Basic properties of prefixes

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

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

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

lemma *Cons-prefix-Cons* [*simp*]: (*x* # *xs* ≤ *y* # *ys*) = (*x* = *y* ∧ *xs* ≤ *ys*)
by (*auto simp add*: *prefix-def*)

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

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

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

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

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

theorem *prefix-Cons*: (*xs* ≤ *y* # *ys*) = (*xs* = [] ∨ (∃ *zs*. *xs* = *y* # *zs* ∧ *zs* ≤ *ys*))
by (*cases xs*) (*auto simp add*: *prefix-def*)

theorem *prefix-append*:
(*xs* ≤ *ys* @ *zs*) = (*xs* ≤ *ys* ∨ (∃ *us*. *xs* = *ys* @ *us* ∧ *us* ≤ *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 ≤ *ys* ==> *length xs* < *length ys* ==> *xs* @ [*ys* ! *length xs*] ≤ *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* ≤ *ys* ==> *length xs* ≤ *length ys*
by (*auto simp add*: *prefix-def*)

lemma *prefix-same-cases*:
(xs_1::'*a list*) ≤ *ys* ⟹ xs_2 ≤ *ys* ⟹ xs_1 ≤ xs_2 ∨ xs_2 ≤ 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
```

6.3 Parallel lists

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

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

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

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

theorem parallel-decomp:
  xs ∥ ys ==> ∃ as b bs c cs. b ≠ c ∧ xs = as @ b # bs ∧ ys = as @ c # cs
proof (induct xs rule: rev-induct)
  case Nil
  then have False by auto
  then show ?case ..
next
  case (snoc x xs)
  show ?case
  proof (rule prefix-cases)
    assume le: xs ≤ ys
    then obtain ys' where ys: ys = xs @ ys' ..
    show ?thesis
    proof (cases ys')
      assume ys' = []
      then show ?thesis by (metis append-Nil2 parallelE prefixI snoc.prems ys)
    next
      fix c cs assume ys': ys' = c # cs
      then show ?thesis
        by (metis Cons-eq-appendI eq-Nil-appendI parallelE prefixI
          same-prefix-prefix snoc.prems ys)
    qed
  next
    assume ys < xs then have ys ≤ xs @ [x] by (simp add: strict-prefix-def)
```

with *snoc* **have** *False* **by** *blast*
then show *?thesis* **..**
next
assume $xs \parallel ys$
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*

6.4 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*⟧ $\Longrightarrow$ *xs* >>= *zs*
by (*auto simp add*: *postfix-def*)
lemma *postfix-antisym*: ⟦*xs* >>= *ys*; *ys* >>= *xs*⟧ $\Longrightarrow$ *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 Myhill-2
```

```
  imports Myhill-1 ~~/src/HOL/Library/List-Prefix
begin
```

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

7.1 Tagging functions

```
definition
   tag-eq :: ('a list ⇒ 'b) ⇒ ('a list × 'a list) set (=-=)
where
   =tag= ≡ {(x, y). tag x = tag y}

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

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

lemma refined-intro:
  assumes ⋀x y z. ⟦x =tag= y; x @ z ∈ A⟧ ⟹ y @ z ∈ A
  shows =tag= ⊆ ≈A
using assms unfolding str-eq-def tag-eq-def
apply(clarify, simp (no-asm-use))
by metis

lemma finite-eq-tag-rel:
  assumes rng-fnt: finite (range tag)
  shows finite (UNIV // =tag=)
proof -
  let ?f = λX. tag ` X and ?A = (UNIV // =tag=)
  have finite (?f ` ?A)
  proof -
    have range ?f ⊆ (Pow (range tag)) unfolding Pow-def by auto
    moreover
    have finite (Pow (range tag)) using rng-fnt by simp
    ultimately
    have finite (range ?f) unfolding image-def by (blast intro: finite-subset)
    moreover
    have ?f ` ?A ⊆ range ?f by auto
    ultimately show finite (?f ` ?A) by (rule rev-finite-subset)
  qed
  moreover
  have inj-on ?f ?A
  proof -
    { fix X Y
```

```
      assume X-in: X ∈ ?A
        and  Y-in: Y ∈ ?A
        and  tag-eq: ?f X = ?f Y
      then obtain x y
        where x ∈ X y ∈ Y tag x = tag y
        unfolding quotient-def Image-def image-def tag-eq-def
        by (simp) (blast)
      with X-in Y-in
      have X = Y
 unfolding quotient-def tag-eq-def by auto
    }
    then show inj-on ?f ?A unfolding inj-on-def by auto
  qed
  ultimately show finite (UNIV // =tag=) by (rule finite-imageD)
qed

lemma refined-partition-finite:
  assumes fnt: finite (UNIV // R1)
  and refined: R1 ⊆ R2
  and eq1: equiv UNIV R1 and eq2: equiv UNIV R2
  shows finite (UNIV // R2)
proof −
  let ?f = λX. {R1 `` {x} | x. x ∈ X}
    and ?A = UNIV // R2 and ?B = UNIV // R1
  have ?f ` ?A ⊆ Pow ?B
    unfolding image-def Pow-def quotient-def by auto
  moreover
  have finite (Pow ?B) using fnt by simp
  ultimately
  have finite (?f ` ?A) by (rule finite-subset)
  moreover
  have inj-on ?f ?A
  proof −
    { fix X Y
      assume X-in: X ∈ ?A and Y-in: Y ∈ ?A and eq-f: ?f X = ?f Y
      from quotientE [OF X-in]
      obtain x where X = R2 `` {x} by blast
      with equiv-class-self[OF eq2] have x-in: x ∈ X by simp
      then have R1 ``{x} ∈ ?f X by auto
      with eq-f have R1 `` {x} ∈ ?f Y by simp
      then obtain y
        where y-in: y ∈ Y and eq-r1-xy: R1 `` {x} = R1 `` {y} by auto
      with eq-equiv-class[OF - eq1]
      have (x, y) ∈ R1 by blast
      with refined have (x, y) ∈ R2 by auto
      with quotient-eqI [OF eq2 X-in Y-in x-in y-in]
      have X = Y .
    }
    then show inj-on ?f ?A unfolding inj-on-def by blast
```

qed
ultimately show *finite* (*UNIV* // *R2*) **by** (*rule finite-imageD*)
qed

lemma *tag-finite-imageD*:
assumes *rng-fnt*: *finite* (*range tag*)
and *refined*: =*tag*= ⊆ ≈*A*
shows *finite* (*UNIV* // ≈*A*)
proof (*rule-tac refined-partition-finite* [*of* =*tag*=])
show *finite* (*UNIV* // =*tag*=) **by** (*rule finite-eq-tag-rel*[*OF rng-fnt*])
next
show =*tag*= ⊆ ≈*A* **using** *refined* **.**
next
show *equiv UNIV* =*tag*=
and *equiv UNIV* (≈*A*)
unfolding *equiv-def str-eq-def tag-eq-def refl-on-def sym-def trans-def*
by *auto*
qed

7.2 Base cases: *Zero*, *One* and *Atom*

lemma *quot-zero-eq*:
shows *UNIV* // ≈{} = {*UNIV*}
unfolding *quotient-def Image-def str-eq-def* **by** *auto*

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

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

lemma *quot-one-finiteI* [*intro*]:
shows *finite* (*UNIV* // ≈{[]})

by (*rule finite-subset*[*OF quot-one-subset*]) (*simp*)

lemma *quot-atom-subset*:
  UNIV // (≈{[*c*]}) ⊆ {{[]},{[*c*]}, *UNIV* − {[], [*c*]}}
proof
  fix *x*
  assume *x* ∈ *UNIV* // ≈{[*c*]}
  then obtain *y* **where** *h*: *x* = {*z*. (*y*, *z*) ∈ ≈{[*c*]}}
    unfolding *quotient-def Image-def* **by** *blast*
  show *x* ∈ {{[]},{[*c*]}, *UNIV* − {[], [*c*]}}
  proof −
    { assume *y* = [] **hence** *x* = {[]} **using** *h*
        by (*auto simp*: *str-eq-def*) **}**
    moreover
    { assume *y* = [*c*] **hence** *x* = {[*c*]} **using** *h*
        by (*auto dest*!: *spec*[**where** *x* = []] *simp*: *str-eq-def*) **}**
    moreover
    { assume *y* ≠ [] **and** *y* ≠ [*c*]
      hence ∀ *z*. (*y* @ *z*) ≠ [*c*] **by** (*case-tac y*, *auto*)
      moreover have ⋀ *p*. (*p* ≠ [] ∧ *p* ≠ [*c*]) = (∀ *q*. *p* @ *q* ≠ [*c*])
        by (*case-tac p*, *auto*)
      ultimately have *x* = *UNIV* − {[],[*c*]} **using** *h*
        by (*auto simp add*: *str-eq-def*)
    }
    ultimately show *?thesis* **by** *blast*
  qed
qed

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

7.3 Case for *Plus*

definition
  tag-Plus :: ′*a lang* ⇒ ′*a lang* ⇒ ′*a list* ⇒ (′*a lang* × ′*a lang*)
where
  tag-Plus A B ≡ λ*x*. (≈*A* ‘‘ {*x*}, ≈*B* ‘‘ {*x*})

lemma *quot-plus-finiteI* [*intro*]:
  assumes *finite1*: *finite* (*UNIV* // ≈*A*)
  and *finite2*: *finite* (*UNIV* // ≈*B*)
  shows *finite* (*UNIV* // ≈(*A* ∪ *B*))
proof (*rule-tac tag* = *tag-Plus A B* **in** *tag-finite-imageD*)
  have *finite* ((*UNIV* // ≈*A*) × (*UNIV* // ≈*B*))
    using *finite1 finite2* **by** *auto*
  then show *finite* (*range* (*tag-Plus A B*))
    unfolding *tag-Plus-def quotient-def*

by (*rule rev-finite-subset*) (*auto*)
next
show =*tag-Plus* A B= $\subseteq$ $\approx(A \cup B)$
unfolding *tag-eq-def tag-Plus-def str-eq-def* **by** *auto*
qed

7.4 Case for *Times*

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

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

lemma *conc-partitions-intro*:
assumes $(u, v) \in$ *Partitions* $x \wedge u \in A \wedge\ v \in B$
shows $x \in A \cdot B$
using *assms* **unfolding** *conc-def Partitions-def*
by *auto*

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

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

lemma *tag-Times-injI*:
assumes a: *tag-Times* A B x = *tag-Times* A B y
and c: $x\ @\ z \in A \cdot B$
shows $y\ @\ z \in A \cdot B$
proof −
from c **obtain** u v **where**
h1: $(u, v) \in$ *Partitions* $(x\ @\ z)$ **and**
h2: $u \in A$ **and**
h3: $v \in B$ **by** (*auto dest*: *conc-partitions-elim*)
from *h1* **have** $x\ @\ z = u\ @\ v$ **unfolding** *Partitions-def* **by** *simp*

then obtain *us*
where $(x = u\ @\ us \wedge us\ @\ z = v) \vee (x\ @\ us = u \wedge z = us\ @\ v)$
by (*auto simp add*: *append-eq-append-conv2*)
moreover
{ assume *eq*: $x = u\ @\ us\ us\ @\ z = v$
have $(\approx B$ ‘‘ $\{us\}) \in$ *snd* (*tag-Times A B x*)
unfolding *Partitions-def tag-Times-def* **using** *h2 eq*
by (*auto simp add*: *str-eq-def*)
then have $(\approx B$ ‘‘ $\{us\}) \in$ *snd* (*tag-Times A B y*)
using *a* **by** *simp*
then obtain u' us' **where**
q1: $u' \in A$ **and**
q2: $\approx B$ ‘‘ $\{us\} = \approx B$ ‘‘ $\{us'\}$ **and**
q3: $(u', us') \in$ *Partitions y*
unfolding *tag-Times-def* **by** *auto*
from *q2 h3 eq*
have $us'\ @\ z \in B$
unfolding *Image-def str-eq-def* **by** *auto*
then have $y\ @\ z \in A \cdot B$ **using** *q1 q3*
unfolding *Partitions-def* **by** *auto*
}
moreover
{ assume *eq*: $x\ @\ us = u\ z = us\ @\ v$
have $(\approx A$ ‘‘ $\{x\}) =$ *fst* (*tag-Times A B x*)
by (*simp add*: *tag-Times-def*)
then have $(\approx A$ ‘‘ $\{x\}) =$ *fst* (*tag-Times A B y*)
using *a* **by** *simp*
then have $\approx A$ ‘‘ $\{x\} = \approx A$ ‘‘ $\{y\}$
by (*simp add*: *tag-Times-def*)
moreover
have $x\ @\ us \in A$ **using** *h2 eq* **by** *simp*
ultimately
have $y\ @\ us \in A$ **using** *equiv-class-member*
unfolding *Image-def str-eq-def* **by** *blast*
then have $(y\ @\ us)\ @\ v \in A \cdot B$
using *h3* **unfolding** *conc-def* **by** *blast*
then have $y\ @\ z \in A \cdot B$ **using** *eq* **by** *simp*
}
ultimately show $y\ @\ z \in A \cdot B$ **by** *blast*
qed

lemma *quot-conc-finiteI* [*intro*]:
assumes *fin1*: *finite* (*UNIV* // $\approx A$)
and *fin2*: *finite* (*UNIV* // $\approx B$)
shows *finite* (*UNIV* // $\approx(A \cdot B)$)
proof (*rule-tac tag* = *tag-Times A B* **in** *tag-finite-imageD*)
have $\bigwedge x\ y\ z.$ ⟦*tag-Times A B x* = *tag-Times A B y*; $x\ @\ z \in A \cdot B$⟧ $\Longrightarrow y\ @\ z \in A \cdot B$
by (*rule tag-Times-injI*)

    (*auto simp add*: *tag-Times-def tag-eq-def*)
  then show $=$*tag-Times A B*$= \subseteq \approx(A \cdot B)$
   by (*rule refined-intro*)
    (*auto simp add*: *tag-eq-def*)
next
  have $*$: *finite* $((UNIV\ //\ \approx A) \times (Pow\ (UNIV\ //\ \approx B)))$
   using *fin1 fin2* **by** *auto*
  show *finite* (*range* (*tag-Times A B*))
   unfolding *tag-Times-def*
   apply(*rule finite-subset*[*OF - **])
   unfolding *quotient-def*
   by *auto*
qed

7.5 Case for *Star*

lemma *star-partitions-elim*:
  assumes $x\ @\ z \in A\star$ $x \neq []$
  shows $\exists (u, v) \in Partitions\ (x\ @\ z).\ u < x \wedge u \in A\star \wedge v \in A\star$
proof $-$
  have $([], x\ @\ z) \in Partitions\ (x\ @\ z)$ $[] < x$ $[] \in A\star$ $x\ @\ z \in A\star$
   using *assms* **by** (*auto simp add*: *Partitions-def strict-prefix-def*)
  then show $\exists (u, v) \in Partitions\ (x\ @\ z).\ u < x \wedge u \in A\star \wedge v \in A\star$
   by *blast*
qed

lemma *finite-set-has-max2*:
  $[\![finite\ A;\ A \neq \{\}]\!] \Longrightarrow \exists\ max \in A.\ \forall\ a \in A.\ length\ a \leq length\ max$
apply(*induct rule*:*finite.induct*)
apply(*simp*)
by (*metis* (*full-types*) *all-not-in-conv insert-iff linorder-linear order-trans*)

lemma *finite-strict-prefix-set*:
  shows *finite* $\{xa.\ xa < (x{::}'a\ list)\}$
apply (*induct x rule*:*rev-induct*, *simp*)
apply (*subgoal-tac* $\{xa.\ xa < xs\ @\ [x]\} = \{xa.\ xa < xs\} \cup \{xs\}$)
by (*auto simp*:*strict-prefix-def*)

lemma *append-eq-cases*:
  assumes a: $x\ @\ y = m\ @\ n$ $m \neq []$
  shows $x \leq m \vee m < x$
unfolding *prefix-def strict-prefix-def* **using** *a*
by (*auto simp add*: *append-eq-append-conv2*)

lemma *star-spartitions-elim2*:
  assumes a: $x\ @\ z \in A\star$
  and b: $x \neq []$
  shows $\exists (u, v) \in Partitions\ x.\ \exists\ (u', v') \in Partitions\ z.\ u < x \wedge u \in A\star \wedge v\ @\ u' \in A \wedge v' \in A\star$

proof −
  def $S \equiv \{u \mid u\ v.\ (u, v) \in Partitions\ x \wedge u < x \wedge u \in A\star \wedge v\ @\ z \in A\star\}$
  have *finite* $\{u.\ u < x\}$ **by** (*rule finite-strict-prefix-set*)
  then have *finite S* **unfolding** *S-def*
    by (*rule rev-finite-subset*) (*auto*)
  moreover
  have $S \neq \{\}$ **using** *a b* **unfolding** *S-def Partitions-def*
    by (*auto simp*: *strict-prefix-def*)
  ultimately have $\exists$ *u-max* $\in S.\ \forall\ u \in S.$ *length u* $\leq$ *length u-max*
    using *finite-set-has-max2* **by** *blast*
  then obtain *u-max v*
    where *h0*: (*u-max*, *v*) $\in$ *Partitions x*
    and *h1*: *u-max* $< x$
    and *h2*: *u-max* $\in A\star$
    and *h3*: $v\ @\ z \in A\star$
    and *h4*: $\forall\ u\ v.\ (u, v) \in Partitions\ x \wedge u < x \wedge u \in A\star \wedge v\ @\ z \in A\star \longrightarrow$ *length u* $\leq$ *length u-max*
    unfolding *S-def Partitions-def* **by** *blast*
  have *q*: $v \neq []$ **using** *h0 h1 b* **unfolding** *Partitions-def* **by** *auto*
  from *h3* **obtain** *a b*
    where *i1*: $(a, b) \in$ *Partitions* $(v\ @\ z)$
    and *i2*: $a \in A$
    and *i3*: $b \in A\star$
    and *i4*: $a \neq []$
    unfolding *Partitions-def*
    using *q* **by** (*auto dest*: *star-decom*)
  have $v \leq a$
  proof (*rule ccontr*)
    assume *a*: $\neg(v \leq a)$
    from *i1* **have** *i1'*: $a\ @\ b = v\ @\ z$ **unfolding** *Partitions-def* **by** *simp*
    then have $a \leq v \vee v < a$ **using** *append-eq-cases q* **by** *blast*
    then have *q*: $a < v$ **using** *a* **unfolding** *strict-prefix-def prefix-def* **by** *auto*
    then obtain *as* **where** *eq*: $a\ @\ as = v$ **unfolding** *strict-prefix-def prefix-def* **by** *auto*
    have (*u-max* @ *a*, *as*) $\in$ *Partitions x* **using** *eq h0* **unfolding** *Partitions-def* **by** *auto*
    moreover
    have *u-max* @ $a < x$ **using** *h0 eq q* **unfolding** *Partitions-def strict-prefix-def prefix-def* **by** *auto*
    moreover
    have *u-max* @ $a \in A\star$ **using** *i2 h2* **by** *simp*
    moreover
    have *as* @ $z \in A\star$ **using** *i1' i2 i3 eq* **by** *auto*
    ultimately have *length* (*u-max* @ *a*) $\leq$ *length u-max* **using** *h4* **by** *blast*
    with *i4* **show** *False* **by** *auto*
  qed
  with *i1* **obtain** *za zb*
    where *k1*: $v\ @\ za = a$
    and *k2*: $(za, zb) \in$ *Partitions z*

```
    and   k4: zb = b
    unfolding Partitions-def prefix-def
    by (auto simp add: append-eq-append-conv2)
  show ∃ (u, v) ∈ Partitions x. ∃ (u', v') ∈ Partitions z. u < x ∧ u ∈ A⋆ ∧ v @
u' ∈ A ∧ v' ∈ A⋆
    using h0 h1 h2 i2 i3 k1 k2 k4 unfolding Partitions-def by blast
qed

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

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

lemma tag-Star-empty-injI:
  assumes a: tag-Star A x = tag-Star A y
  and     c: x @ z ∈ A⋆
  and     d: x = []
  shows y @ z ∈ A⋆
proof −
  from a have {} = tag-Star A y unfolding tag-Star-def using d by auto
```

```
  then have y = []
    unfolding tag-Star-def Partitions-def strict-prefix-def prefix-def
    by (auto) (metis Nil-in-star append-self-conv2)
  then show y @ z ∈ A⋆ using c d by simp
qed

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

7.6 The conclusion of the second direction

```
lemma Myhill-Nerode2:
  fixes r::'a rexp
  shows finite (UNIV // ≈(lang r))
by (induct r) (auto)

end
```

8 Derivatives of regular expressions

```
theory Derivatives
imports Regular-Exp
begin
```

This theory is based on work by Brozowski [?] and Antimirov [?].

8.1 Left-Quotients of languages

```
definition Deriv :: 'a ⇒ 'a lang ⇒ 'a lang
where Deriv x A = { xs. x#xs ∈ A }

definition Derivs :: 'a list ⇒ 'a lang ⇒ 'a lang
where Derivs xs A = { ys. xs @ ys ∈ A }

abbreviation
```

```
  Derivss :: 'a list ⇒ 'a lang set ⇒ 'a lang
where
  Derivss s As ≡ ⋃ (Derivs s) ` As


lemma Deriv-empty[simp]:   Deriv a {} = {}
  and Deriv-epsilon[simp]: Deriv a {[]} = {}
  and Deriv-char[simp]:    Deriv a {[b]} = (if a = b then {[]} else {})
  and Deriv-union[simp]:   Deriv a (A ∪ B) = Deriv a A ∪ Deriv a B
by (auto simp: Deriv-def)

lemma Deriv-conc-subset:
Deriv a A @@ B ⊆ Deriv a (A @@ B) (is ?L ⊆ ?R)
proof
  fix w assume w ∈ ?L
  then obtain u v where w = u @ v a # u ∈ A v ∈ B
    by (auto simp: Deriv-def)
  then have a # w ∈ A @@ B
    by (auto intro: concI[of a # u, simplified])
  thus w ∈ ?R by (auto simp: Deriv-def)
qed

lemma Der-conc [simp]:
  shows Deriv c (A @@ B) = (Deriv c A) @@ B ∪ (if [] ∈ A then Deriv c B else
{})
unfolding Deriv-def conc-def
by (auto simp add: Cons-eq-append-conv)

lemma Deriv-star [simp]:
  shows Deriv c (star A) = (Deriv c A) @@ star A
proof −
  have incl: [] ∈ A ⟹ Deriv c (star A) ⊆ (Deriv c A) @@ star A
    unfolding Deriv-def conc-def
    apply(auto simp add: Cons-eq-append-conv)
    apply(drule star-decom)
    apply(auto simp add: Cons-eq-append-conv)
    done

  have Deriv c (star A) = Deriv c (A @@ star A ∪ {[]})
    by (simp only: star-unfold-left[symmetric])
  also have ... = Deriv c (A @@ star A)
    by (simp only: Deriv-union) (simp)
  also have ... = (Deriv c A) @@ (star A) ∪ (if [] ∈ A then Deriv c (star A) else
{})
    by simp
   also have ... = (Deriv c A) @@ star A
    using incl by auto
  finally show Deriv c (star A) = (Deriv c A) @@ star A .
qed
```

lemma *Derivs-simps* [*simp*]:
shows *Derivs* [] *A* = *A*
and *Derivs* (*c* # *s*) *A* = *Derivs s* (*Deriv c A*)
and *Derivs* (*s1* @ *s2*) *A* = *Derivs s2* (*Derivs s1 A*)
unfolding *Derivs-def Deriv-def* **by** *auto*

8.2 Brozowski's derivatives of regular expressions

```
fun
  nullable :: 'a rexp ⇒ bool
where
  nullable (Zero) = False
| nullable (One) = True
| nullable (Atom c) = False
| nullable (Plus r1 r2) = (nullable r1 ∨ nullable r2)
| nullable (Times r1 r2) = (nullable r1 ∧ nullable r2)
| nullable (Star r) = True

fun
  deriv :: 'a ⇒ 'a rexp ⇒ 'a rexp
where
  deriv c (Zero) = Zero
| deriv c (One) = Zero
| deriv c (Atom c') = (if c = c' then One else Zero)
| deriv c (Plus r1 r2) = Plus (deriv c r1) (deriv c r2)
| deriv c (Times r1 r2) =
    (if nullable r1 then Plus (Times (deriv c r1) r2) (deriv c r2) else Times (deriv
c r1) r2)
| deriv c (Star r) = Times (deriv c r) (Star r)

fun
  derivs :: 'a list ⇒ 'a rexp ⇒ 'a rexp
where
  derivs [] r = r
| derivs (c # s) r = derivs s (deriv c r)
```

lemma *nullable-iff*:
shows *nullable r* ⟷ [] ∈ *lang r*
by (*induct r*) (*auto simp add*: *conc-def split*: *if-splits*)

lemma *Deriv-deriv*:
shows *Deriv c* (*lang r*) = *lang* (*deriv c r*)
by (*induct r*) (*simp-all add*: *nullable-iff*)

lemma *Derivs-derivs*:
shows *Derivs s* (*lang r*) = *lang* (*derivs s r*)
by (*induct s arbitrary*: *r*) (*simp-all add*: *Deriv-deriv*)

8.3 Antimirov's partial derivatives

abbreviation
Timess rs r ≡ {*Times r′ r* | *r′*. *r′* ∈ *rs*}

fun
pderiv :: ′*a* ⇒ ′*a rexp* ⇒ ′*a rexp set*
where
pderiv c Zero = {}
| *pderiv c One* = {}
| *pderiv c* (*Atom c′*) = (*if c* = *c′ then* {*One*} *else* {})
| *pderiv c* (*Plus r1 r2*) = (*pderiv c r1*) ∪ (*pderiv c r2*)
| *pderiv c* (*Times r1 r2*) =
(*if nullable r1 then Timess* (*pderiv c r1*) *r2* ∪ *pderiv c r2 else Timess* (*pderiv c r1*) *r2*)
| *pderiv c* (*Star r*) = *Timess* (*pderiv c r*) (*Star r*)

fun
pderivs :: ′*a list* ⇒ ′*a rexp* ⇒ (′*a rexp*) *set*
where
pderivs [] *r* = {*r*}
| *pderivs* (*c* # *s*) *r* = ⋃ (*pderivs s*) ‘ (*pderiv c r*)

abbreviation
pderiv-set :: ′*a* ⇒ ′*a rexp set* ⇒ ′*a rexp set*
where
pderiv-set c rs ≡ ⋃ *pderiv c* ‘ *rs*

abbreviation
pderivs-set :: ′*a list* ⇒ ′*a rexp set* ⇒ ′*a rexp set*
where
pderivs-set s rs ≡ ⋃ (*pderivs s*) ‘ *rs*

lemma *pderivs-append*:
pderivs (*s1* @ *s2*) *r* = ⋃ (*pderivs s2*) ‘ (*pderivs s1 r*)
by (*induct s1 arbitrary*: *r*) (*simp-all*)

lemma *pderivs-snoc*:
shows *pderivs* (*s* @ [*c*]) *r* = *pderiv-set c* (*pderivs s r*)
by (*simp add*: *pderivs-append*)

lemma *pderivs-simps* [*simp*]:
shows *pderivs s Zero* = (*if s* = [] *then* {*Zero*} *else* {})
and *pderivs s One* = (*if s* = [] *then* {*One*} *else* {})
and *pderivs s* (*Plus r1 r2*) = (*if s* = [] *then* {*Plus r1 r2*} *else* (*pderivs s r1*) ∪ (*pderivs s r2*))
by (*induct s*) (*simp-all*)

lemma *pderivs-Atom*:
shows *pderivs s* (*Atom c*) ⊆ {*Atom c*, *One*}

by (*induct s*) (*simp-all*)

8.4 Relating left-quotients and partial derivatives

lemma *Deriv-pderiv*:
  shows $Deriv\ c\ (lang\ r) = \bigcup\ lang\ `\ (pderiv\ c\ r)$
by (*induct r*) (*auto simp add*: *nullable-iff conc-UNION-distrib*)

lemma *Derivs-pderivs*:
  shows $Derivs\ s\ (lang\ r) = \bigcup\ lang\ `\ (pderivs\ s\ r)$
proof (*induct s arbitrary*: *r*)
  case (*Cons c s*)
  have *ih*: $\bigwedge r.\ Derivs\ s\ (lang\ r) = \bigcup\ lang\ `\ (pderivs\ s\ r)$ **by** *fact*
  have $Derivs\ (c\ \#\ s)\ (lang\ r) = Derivs\ s\ (Deriv\ c\ (lang\ r))$ **by** *simp*
  also have $\ldots = Derivs\ s\ (\bigcup\ lang\ `\ (pderiv\ c\ r))$ **by** (*simp add*: *Deriv-pderiv*)
  also have $\ldots = Derivss\ s\ (lang\ `\ (pderiv\ c\ r))$
    by (*auto simp add*: *Derivs-def*)
  also have $\ldots = \bigcup\ lang\ `\ (pderivs\text{-}set\ s\ (pderiv\ c\ r))$
    using *ih* **by** *auto*
  also have $\ldots = \bigcup\ lang\ `\ (pderivs\ (c\ \#\ s)\ r)$ **by** *simp*
  finally show $Derivs\ (c\ \#\ s)\ (lang\ r) = \bigcup\ lang\ `\ pderivs\ (c\ \#\ s)\ r$ **.**
qed (*simp add*: *Derivs-def*)

8.5 Relating derivatives and partial derivatives

lemma *deriv-pderiv*:
  shows $(\bigcup\ lang\ `\ (pderiv\ c\ r)) = lang\ (deriv\ c\ r)$
unfolding *Deriv-deriv*[*symmetric*] *Deriv-pderiv* **by** *simp*

lemma *derivs-pderivs*:
  shows $(\bigcup\ lang\ `\ (pderivs\ s\ r)) = lang\ (derivs\ s\ r)$
unfolding *Derivs-derivs*[*symmetric*] *Derivs-pderivs* **by** *simp*

8.6 Finiteness property of partial derivatives

definition
  $pderivs\text{-}lang :: 'a\ lang \Rightarrow 'a\ rexp \Rightarrow 'a\ rexp\ set$
where
  $pderivs\text{-}lang\ A\ r \equiv \bigcup x \in A.\ pderivs\ x\ r$

lemma *pderivs-lang-subsetI*:
  assumes $\bigwedge s.\ s \in A \Longrightarrow pderivs\ s\ r \subseteq C$
  shows $pderivs\text{-}lang\ A\ r \subseteq C$
using *assms* **unfolding** *pderivs-lang-def* **by** (*rule UN-least*)

lemma *pderivs-lang-union*:
  shows $pderivs\text{-}lang\ (A \cup B)\ r = (pderivs\text{-}lang\ A\ r \cup pderivs\text{-}lang\ B\ r)$
by (*simp add*: *pderivs-lang-def*)

lemma *pderivs-lang-subset*:

shows $A \subseteq B \Longrightarrow$ *pderivs-lang* A $r \subseteq$ *pderivs-lang* B r
by (*auto simp add*: *pderivs-lang-def*)

definition
UNIV1 $\equiv$ *UNIV* $-$ $\{[]\}$

lemma *pderivs-lang-Zero* [*simp*]:
shows *pderivs-lang UNIV1 Zero* = {}
unfolding *UNIV1-def pderivs-lang-def* **by** *auto*

lemma *pderivs-lang-One* [*simp*]:
shows *pderivs-lang UNIV1 One* = {}
unfolding *UNIV1-def pderivs-lang-def* **by** (*auto split*: *if-splits*)

lemma *pderivs-lang-Atom* [*simp*]:
shows *pderivs-lang UNIV1* (*Atom c*) = {*One*}
unfolding *UNIV1-def pderivs-lang-def*
apply(*auto*)
apply(*frule rev-subsetD*)
apply(*rule pderivs-Atom*)
apply(*simp*)
apply(*case-tac xa*)
apply(*auto split*: *if-splits*)
done

lemma *pderivs-lang-Plus* [*simp*]:
shows *pderivs-lang UNIV1* (*Plus r1 r2*) = *pderivs-lang UNIV1 r1* $\cup$ *pderivs-lang UNIV1 r2*
unfolding *UNIV1-def pderivs-lang-def* **by** *auto*

Non-empty suffixes of a string (needed for the cases of *Times* and *Star* below)

definition
PSuf s $\equiv$ $\{v.\ v \neq [] \wedge (\exists\, u.\ u\ @\ v = s)\}$

lemma *PSuf-snoc*:
shows *PSuf* (s @ $[c]$) = (*PSuf s*) @@ $\{[c]\} \cup \{[c]\}$
unfolding *PSuf-def conc-def*
by (*auto simp add*: *append-eq-append-conv2 append-eq-Cons-conv*)

lemma *PSuf-Union*:
shows $(\bigcup v \in$ *PSuf s* @@ $\{[c]\}.\ f\ v) = (\bigcup v \in$ *PSuf s*. $f\ (v\ @\ [c]))$
by (*auto simp add*: *conc-def*)

lemma *pderivs-lang-snoc*:
shows *pderivs-lang* (*PSuf s* @@ $\{[c]\}$) r = (*pderiv-set c* (*pderivs-lang* (*PSuf s*) r))
unfolding *pderivs-lang-def*
by (*simp add*: *PSuf-Union pderivs-snoc*)

lemma *pderivs-Times*:
shows *pderivs s* (*Times r1 r2*) ⊆ *Timess* (*pderivs s r1*) *r2* ∪ (*pderivs-lang* (*PSuf s*) *r2*)
proof (*induct s rule*: *rev-induct*)
case (*snoc c s*)
have *ih*: *pderivs s* (*Times r1 r2*) ⊆ *Timess* (*pderivs s r1*) *r2* ∪ (*pderivs-lang* (*PSuf s*) *r2*)
by *fact*
have *pderivs* (*s* @ [*c*]) (*Times r1 r2*) = *pderiv-set c* (*pderivs s* (*Times r1 r2*))
by (*simp add*: *pderivs-snoc*)
also have ... ⊆ *pderiv-set c* (*Timess* (*pderivs s r1*) *r2* ∪ (*pderivs-lang* (*PSuf s*) *r2*))
using *ih* **by** (*auto*) (*blast*)
also have ... = *pderiv-set c* (*Timess* (*pderivs s r1*) *r2*) ∪ *pderiv-set c* (*pderivs-lang* (*PSuf s*) *r2*)
by (*simp*)
also have ... = *pderiv-set c* (*Timess* (*pderivs s r1*) *r2*) ∪ *pderivs-lang* (*PSuf s* @@ {[*c*]}) *r2*
by (*simp add*: *pderivs-lang-snoc*)
also
have ... ⊆ *pderiv-set c* (*Timess* (*pderivs s r1*) *r2*) ∪ *pderiv c r2* ∪ *pderivs-lang* (*PSuf s* @@ {[*c*]}) *r2*
by *auto*
also
have ... ⊆ *Timess* (*pderiv-set c* (*pderivs s r1*)) *r2* ∪ *pderiv c r2* ∪ *pderivs-lang* (*PSuf s* @@ {[*c*]}) *r2*
by (*auto simp add*: *if-splits*) (*blast*)
also have ... = *Timess* (*pderivs* (*s* @ [*c*]) *r1*) *r2* ∪ *pderiv c r2* ∪ *pderivs-lang* (*PSuf s* @@ {[*c*]}) *r2*
by (*simp add*: *pderivs-snoc*)
also have ... ⊆ *Timess* (*pderivs* (*s* @ [*c*]) *r1*) *r2* ∪ *pderivs-lang* (*PSuf* (*s* @ [*c*])) *r2*
unfolding *pderivs-lang-def* **by** (*auto simp add*: *PSuf-snoc*)
finally show *?case* **.**
qed (*simp*)

lemma *pderivs-lang-Times-aux1*:
assumes *a*: *s* ∈ *UNIV1*
shows *pderivs-lang* (*PSuf s*) *r* ⊆ *pderivs-lang UNIV1 r*
using *a* **unfolding** *UNIV1-def PSuf-def pderivs-lang-def* **by** *auto*

lemma *pderivs-lang-Times-aux2*:
assumes *a*: *s* ∈ *UNIV1*
shows *Timess* (*pderivs s r1*) *r2* ⊆ *Timess* (*pderivs-lang UNIV1 r1*) *r2*
using *a* **unfolding** *pderivs-lang-def* **by** *auto*

lemma *pderivs-lang-Times*:
shows *pderivs-lang UNIV1* (*Times r1 r2*) ⊆ *Timess* (*pderivs-lang UNIV1 r1*)

r2 ∪ *pderivs-lang UNIV1 r2*
apply(*rule pderivs-lang-subsetI*)
apply(*rule subset-trans*)
apply(*rule pderivs-Times*)
using *pderivs-lang-Times-aux1 pderivs-lang-Times-aux2*
apply(*blast*)
done

lemma *pderivs-Star*:
  assumes *a*: $s \neq []$
  shows *pderivs s* (*Star r*) ⊆ *Timess* (*pderivs-lang* (*PSuf s*) *r*) (*Star r*)
using *a*
proof (*induct s rule*: *rev-induct*)
  case (*snoc c s*)
  have *ih*: $s \neq [] \Longrightarrow$ *pderivs s* (*Star r*) ⊆ *Timess* (*pderivs-lang* (*PSuf s*) *r*) (*Star r*) **by** *fact*
  { assume *asm*: $s \neq []$
    have *pderivs* (*s* @ [*c*]) (*Star r*) = *pderiv-set c* (*pderivs s* (*Star r*)) **by** (*simp add*: *pderivs-snoc*)
    also have ... ⊆ *pderiv-set c* (*Timess* (*pderivs-lang* (*PSuf s*) *r*) (*Star r*))
      using *ih*[*OF asm*] **by** (*auto*) (*blast*)
    also have ... ⊆ *Timess* (*pderiv-set c* (*pderivs-lang* (*PSuf s*) *r*)) (*Star r*) ∪ *pderiv c* (*Star r*)
      by (*auto split*: *if-splits*) (*blast*)+
    also have ... ⊆ *Timess* (*pderivs-lang* (*PSuf* (*s* @ [*c*])) *r*) (*Star r*) ∪ (*Timess* (*pderiv c r*) (*Star r*))
      by (*simp only*: *PSuf-snoc pderivs-lang-snoc pderivs-lang-union*)
        (*auto simp add*: *pderivs-lang-def*)
    also have ... = *Timess* (*pderivs-lang* (*PSuf* (*s* @ [*c*])) *r*) (*Star r*)
      by (*auto simp add*: *PSuf-snoc PSuf-Union pderivs-snoc pderivs-lang-def*)
    finally have *?case* **.**
  }
  moreover
  { assume *asm*: *s* = []
    then have *?case*
      apply (*auto simp add*: *pderivs-lang-def pderivs-snoc PSuf-def*)
      apply(*rule-tac x* = [*c*] **in** *exI*)
      apply(*auto*)
      done
  }
  ultimately show *?case* **by** *blast*
qed (*simp*)

lemma *pderivs-lang-Star*:
  shows *pderivs-lang UNIV1* (*Star r*) ⊆ *Timess* (*pderivs-lang UNIV1 r*) (*Star r*)
apply(*rule pderivs-lang-subsetI*)
apply(*rule subset-trans*)
apply(*rule pderivs-Star*)
apply(*simp add*: *UNIV1-def*)

apply(*simp add*: *UNIV1-def PSuf-def*)
apply(*auto simp add*: *pderivs-lang-def*)
done

lemma *finite-Timess* [*simp*]:
  assumes *a*: *finite A*
  shows *finite* (*Timess A r*)
using *a* **by** *auto*

lemma *finite-pderivs-lang-UNIV1*:
  shows *finite* (*pderivs-lang UNIV1 r*)
apply(*induct r*)
apply(*simp-all add*:
  finite-subset[*OF pderivs-lang-Times*]
  finite-subset[*OF pderivs-lang-Star*])
done

lemma *pderivs-lang-UNIV*:
  shows *pderivs-lang UNIV r* = *pderivs* [] *r* ∪ *pderivs-lang UNIV1 r*
unfolding *UNIV1-def pderivs-lang-def*
by *blast*

lemma *finite-pderivs-lang-UNIV*:
  shows *finite* (*pderivs-lang UNIV r*)
unfolding *pderivs-lang-UNIV*
by (*simp add*: *finite-pderivs-lang-UNIV1*)

lemma *finite-pderivs-lang*:
  shows *finite* (*pderivs-lang A r*)
by (*metis finite-pderivs-lang-UNIV pderivs-lang-subset rev-finite-subset subset-UNIV*)

8.7 A regular expression matcher based on Brozowski's derivatives

fun
  matcher :: *'a rexp* ⇒ *'a list* ⇒ *bool*
where
  matcher r s = *nullable* (*derivs s r*)

lemma *matcher-correctness*:
  shows *matcher r s* ⟷ *s* ∈ *lang r*
by (*induct s arbitrary*: *r*)
   (*simp-all add*: *nullable-iff Deriv-deriv*[*symmetric*] *Deriv-def*)

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

9 The theorem

```
theorem Myhill-Nerode:
  fixes A::('a::finite) lang
  shows (∃ r. A = lang r) ⟷ finite (UNIV // ≈A)
using Myhill-Nerode1 Myhill-Nerode2 by auto
```

9.1 Second direction proved using partial derivatives

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

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

lemma Myhill-Nerode3:
  fixes r::'a rexp
  shows finite (UNIV // ≈(lang r))
proof -
  have finite (UNIV // =(λx. pderivs x r)=)
  proof -
    have range (λx. pderivs x r) ⊆ Pow (pderivs-lang UNIV r)
      unfolding pderivs-lang-def by auto
    moreover
    have finite (Pow (pderivs-lang UNIV r)) by (simp add: finite-pderivs-lang)
    ultimately
    have finite (range (λx. pderivs x r))
      by (simp add: finite-subset)
    then show finite (UNIV // =(λx. pderivs x r)=)
      by (rule finite-eq-tag-rel)
  qed
  moreover
  have =(λx. pderivs x r)= ⊆ ≈(lang r)
    unfolding tag-eq-def
    by (auto simp add: MN-Rel-Derivs Derivs-pderivs)
  moreover
  have equiv UNIV =(λx. pderivs x r)=
  and  equiv UNIV (≈(lang r))
    unfolding equiv-def refl-on-def sym-def trans-def
    unfolding tag-eq-def str-eq-def
    by auto
  ultimately show finite (UNIV // ≈(lang r))
    by (rule refined-partition-finite)
qed

end
```