Combinatorics on Words formalized
Basics

Štěpán Holub
Martin Raška
Štěpán Starosta

December 14, 2021

Funded by the Czech Science Foundation grant GAČR 20-20621S.

Contents

theory *Arithmetical-Hints*
imports *Main*
begin

0.1 Arithmetical hints

In this section we give some specific auxiliary lemmas on natural integers.

lemma *zero-diff-eq*: $i \leq j \Longrightarrow (0::nat) = j - i \Longrightarrow j = i$
by *simp*

lemma *zero-less-diff'*: $i < j \Longrightarrow j - i \neq (0::nat)$
by *simp*

lemma *nat-prod-le*: $m \neq (0 :: nat) \Longrightarrow m*n \leq k \Longrightarrow n \leq k$
using *le-trans*[*of n m*∗*n k*] **by** *auto*

lemma *get-div*: $(p :: nat) < a \Longrightarrow m = (m * a + p)\ div\ a$
by *simp*

lemma *get-mod*: $(p :: nat) < a \Longrightarrow p = (m * a + p)\ mod\ a$
by *simp*

lemma *plus-one-between*: $(a :: nat) < b \Longrightarrow \neg\ b < a + 1$
by *auto*

lemma *quotient-smaller*: $a \neq 0 \Longrightarrow a = k * b \Longrightarrow b \leq (a::nat)$
by *simp*

lemma *mult-cancel-le*: $b \neq 0 \Longrightarrow a*b \leq c*b \Longrightarrow a \leq (c::nat)$
by *simp*

lemma *add-lessD2*: **assumes** $k + m < (n::nat)$ **shows** $m < n$
using *add-lessD1*[*OF assms*[*unfolded add.commute*[*of k m*]]]**.**

lemma *mod-offset*: **assumes** $M \neq (0 :: nat)$
obtains k **where** $n\ mod\ M = (l + k)\ mod\ M$
proof−
have $(l + (M - l\ mod\ M))\ mod\ M = 0$
using *mod-add-left-eq*[*of l M* $(M - l\ mod\ M)$, *unfolded le-add-diff-inverse*[*OF mod-le-divisor*[*OF assms*[*unfolded neq0-conv*]], *of l*] *mod-self*, *symmetric*]**.**
from *mod-add-left-eq*[*of* $(l + (M - l\ mod\ M))\ M\ n$, *symmetric*, *unfolded this add.commute*[*of 0*] *add.comm-neutral*]
have $((l + (M - l\ mod\ M)) + n)\ mod\ M = n\ mod\ M$**.**
from *that*[*OF this*[*unfolded add.assoc*, *symmetric*]]

show *thesis*.
qed

lemma assumes $q \neq (0::nat)$ **shows** $p \leq p + q - gcd\ p\ q$
using *gcd-le2-nat*[*OF* ‹$q \neq 0$›, *of p*]
by *linarith*

lemma *less-mult-one*: **assumes** $(m-1)*k < k$ **obtains** $m = 0 \mid m = (1::nat)$
using *assms* **by** *fastforce*

end

theory *Reverse-Symmetry*
imports *Main*
begin

Chapter 1

Reverse symmetry

This theory deals with a mechanism which produces new facts on lists from already known facts by the reverse symmetry of lists, induced by the mapping *rev*. It constructs the rule attribute "reversed" which produces the symmetrical fact using so-called reversal rules, which are rewriting rules that may be applied to obtain the symmetrical fact. An example of such a reversal rule is the already existing *rev ys* @ *rev xs* = *rev* (*xs* @ *ys*). Some additional reversal rules are given in this theory.

The symmetrical fact 'A[reversed]' is constructed from the fact 'A' in the following manner: 1. each schematic variable *xs* of type *'a list* is instantiated by *rev xs*; 2. constant Nil is substituted by *rev* []; 3. each quantification of *'a list* type variable $\bigwedge xs.\ P\ xs$ is substituted by (logically equivalent) quantification $\bigwedge xs.\ P\ (rev\ xs)$, similarly for $\forall$, $\exists$ and $\exists!$ quantifiers; each bounded quantification of *'a list* type variable $\forall\, xs{\in}A.\ P\ xs$ is substituted by (logically equivalent) quantification $\forall\, xs{\in}rev\ `\ A.\ P\ (rev\ xs)$, similarly for bounded $\exists$ quantifier; 4. simultaneous rewrites according to a the current list of reversal rules are performed; 5. final correctional rewrites related to reversion of (#) are performed.

List of reversal rules is maintained by declaration attribute "reversal_rule" with standard "add" and "del" options.

See examples at the end of the file.

1.1 Quantifications and maps

lemma *all-surj-conv*: **assumes** *surj f* **shows** $(\bigwedge x.\ PROP\ P\ (f\ x)) \equiv (\bigwedge y.\ PROP\ P\ y)$
proof
  fix *y* **assume** $\bigwedge x.\ PROP\ P\ (f\ x)$
  then have *PROP P* (*f* (*inv f y*))**.**
  then show *PROP P y* **unfolding** *surj-f-inv-f*[*OF assms*]**.**
qed

lemma *All-surj-conv*: **assumes** *surj f* **shows** $(\forall x.\ P\ (f\ x)) \longleftrightarrow (\forall y.\ P\ y)$
proof (*intro iffI allI*)
  fix *y* **assume** $\forall x.\ P\ (f\ x)$
  then have $P\ (f\ (inv\ f\ y))$**..**
  then show $P\ y$ **unfolding** *surj-f-inv-f*[*OF assms*]**.**
qed *simp*

lemma *Ex-surj-conv*: **assumes** *surj f* **shows** $(\exists x.\ P\ (f\ x)) \longleftrightarrow (\exists y.\ P\ y)$
proof
  assume $\exists x.\ P\ (f\ x)$
  then obtain *x* **where** $P\ (f\ x)$**..**
  then show $\exists x.\ P\ x$**..**
next
  assume $\exists y.\ P\ y$
  then obtain *y* **where** $P\ y$**..**
  then have $P\ (f\ (inv\ f\ y))$ **unfolding** *surj-f-inv-f*[*OF assms*]**.**
  then show $\exists x.\ P\ (f\ x)$**..**
qed

lemma *Ex1-bij-conv*: **assumes** *bij f* **shows** $(\exists!x.\ P\ (f\ x)) \longleftrightarrow (\exists!y.\ P\ y)$
proof
  have *imp*: $\exists!y.\ Q\ y$ **if** *bij*: *bij g* **and** *ex1*: $\exists!x.\ Q\ (g\ x)$ **for** *g Q*
  proof −
    from *ex1E*[*OF ex1*, *rule-format*]
    obtain *x* **where** *ex*: $Q\ (g\ x)$ **and** *uniq*: $\bigwedge x'.\ Q\ (g\ x') \Longrightarrow x' = x$ **by** *blast*
    {
      fix *y* **assume** *Q y*
      then have $Q\ (g\ (inv\ g\ y))$ **unfolding** *surj-f-inv-f*[*OF bij-is-surj*[*OF bij*]]**.**
      from *uniq*[*OF this*] **have** $x = inv\ g\ y$**..**
      then have $y = g\ x$ **unfolding** *bij-inv-eq-iff*[*OF bij*]**..**
    }
    with *ex* **show** $\exists!y.\ Q\ y$**..**
  qed
  show $\exists!x.\ P\ (f\ x) \Longrightarrow \exists!y.\ P\ y$ **using** *imp*[*OF assms*]**.**
  assume $\exists!y.\ P\ y$
  then have $\exists!y.\ P\ (f\ (inv\ f\ y))$ **unfolding** *surj-f-inv-f*[*OF bij-is-surj*[*OF assms*]]**.**
  from *imp*[*OF bij-imp-bij-inv*[*OF assms*] *this*]
  show $\exists!x.\ P\ (f\ x)$**.**
qed

lemma *Ball-inj-conv*: **assumes** *inj f* **shows** $(\forall y{\in}f\ `\ A.\ P\ (inv\ f\ y)) \longleftrightarrow (\forall x{\in}A.\ P\ x)$
  using *ball-simps*(*9*)[*of f A* $\lambda y.\ P\ (inv\ f\ y)$] **unfolding** *inv-f-f*[*OF assms*]**.**

lemma *Bex-inj-conv*: **assumes** *inj f* **shows** $(\exists y{\in}f\ `\ A.\ P\ (inv\ f\ y)) \longleftrightarrow (\exists x{\in}A.\ P\ x)$
  using *bex-simps*(*7*)[*of f A* $\lambda y.\ P\ (inv\ f\ y)$] **unfolding** *inv-f-f*[*OF assms*]**.**

1.1.1 Quantifications and reverse

lemma *rev-involution*: $rev \circ rev = id$
by *auto*

lemma *rev-bij*: *bij rev*
using *o-bij*[*OF rev-involution rev-involution*].

lemma *rev-inv*: *inv rev* = *rev*
using *inv-unique-comp*[*OF rev-involution rev-involution*].

lemmas *all-rev-conv* = *all-surj-conv*[*OF bij-is-surj*[*OF rev-bij*]]
and *All-rev-conv* = *All-surj-conv*[*OF bij-is-surj*[*OF rev-bij*]]
and *Ex-rev-conv* = *Ex-surj-conv*[*OF bij-is-surj*[*OF rev-bij*]]
and *Ex1-rev-conv* = *Ex1-bij-conv*[*OF rev-bij*]
and *Ball-rev-conv* = *Ball-inj-conv*[*OF bij-is-inj*[*OF rev-bij*], *unfolded rev-inv*]
and *Bex-rev-conv* = *Bex-inj-conv*[*OF bij-is-inj*[*OF rev-bij*], *unfolded rev-inv*]

1.2 Attributes

context
begin

1.2.1 Definitons of reverse wrapers

private definition *rev-Nil-wrap* :: *'a list*
where *rev-Nil-wrap* = *rev Nil*

private definition *all-rev-wrap* :: $('a\ list \Rightarrow prop) \Rightarrow prop$
where *all-rev-wrap P* $\equiv (\bigwedge x.\ PROP\ P\ (rev\ x))$

private definition *All-rev-wrap* :: $('a\ list \Rightarrow bool) \Rightarrow bool$
where *All-rev-wrap P* $= (\forall x.\ P\ (rev\ x))$

private definition *Ex-rev-wrap* :: $('a\ list \Rightarrow bool) \Rightarrow bool$
where *Ex-rev-wrap P* $= (\exists x.\ P\ (rev\ x))$

private definition *Ex1-rev-wrap* :: $('a\ list \Rightarrow bool) \Rightarrow bool$
where *Ex1-rev-wrap P* $= (\exists !x.\ P\ (rev\ x))$

private definition *Ball-rev-wrap* :: $'a\ list\ set \Rightarrow ('a\ list \Rightarrow bool) \Rightarrow bool$
where *Ball-rev-wrap A P* $= (\forall x \in rev\ `\ A.\ P\ (rev\ x))$

private definition *Bex-rev-wrap* :: $'a\ list\ set \Rightarrow ('a\ list \Rightarrow bool) \Rightarrow bool$
where *Bex-rev-wrap A P* $= (\exists x \in rev\ `\ A.\ P\ (rev\ x))$

1.2.2 Initial reversal rules

private lemma *rev-Nil*: *rev Nil* = *Nil*
by *simp*

private lemmas *init-rev-wrap* =
eq-reflection[*OF trans*[*OF rev-Nil*[*symmetric*] *rev-Nil-wrap-def*[*symmetric*]]]
transitive[*OF all-rev-conv*[*symmetric*] *all-rev-wrap-def*[*symmetric*], *of P P* **for** *P*]
eq-reflection[*OF trans*[*OF All-rev-conv*[*symmetric*] *All-rev-wrap-def*[*symmetric*], *of P P* **for** *P*]]
eq-reflection[*OF trans*[*OF Ex-rev-conv*[*symmetric*] *Ex-rev-wrap-def*[*symmetric*], *of P P* **for** *P*]]
eq-reflection[*OF trans*[*OF Ex1-rev-conv*[*symmetric*] *Ex1-rev-wrap-def*[*symmetric*], *of P P* **for** *P*]]
eq-reflection[*OF trans*[*OF Ball-rev-conv*[*symmetric*] *Ball-rev-wrap-def*[*symmetric*], *of P P* **for** *P*]]
eq-reflection[*OF trans*[*OF Bex-rev-conv*[*symmetric*] *Bex-rev-wrap-def*[*symmetric*], *of P P* **for** *P*]]

private lemma *all-rev-unwrap*: *all-rev-wrap* ($\lambda x.$ *PROP P x*) $\equiv$ ($\bigwedge x.$ *PROP P* (*rev x*))
using *all-rev-wrap-def*.

private lemma *All-rev-unwrap*: *All-rev-wrap* ($\lambda x.$ *P x*) = ($\forall x.$ *P* (*rev x*))
using *All-rev-wrap-def*.

private lemma *Ex-rev-unwrap*: *Ex-rev-wrap* ($\lambda x.$ *P x*) = ($\exists x.$ *P* (*rev x*))
using *Ex-rev-wrap-def*.

private lemma *Ex1-rev-unwrap*: *Ex1-rev-wrap* ($\lambda x.$ *P x*) = ($\exists! x.$ *P* (*rev x*))
using *Ex1-rev-wrap-def*.

private lemma *Ball-rev-unwrap*: *Ball-rev-wrap A* ($\lambda x.$ *P x*) = ($\forall x \in$ *rev* ‘ *A*. *P* (*rev x*))
using *Ball-rev-wrap-def*.

private lemma *Bex-rev-unwrap*: *Bex-rev-wrap A* ($\lambda x.$ *P x*) = ($\exists x \in$ *rev* ‘ *A*. *P* (*rev x*))
using *Bex-rev-wrap-def*.

private lemmas *init-rev-unwrap* =
eq-reflection[*OF rev-Nil-wrap-def*]
all-rev-unwrap
eq-reflection[*OF All-rev-unwrap*]
eq-reflection[*OF Ex-rev-unwrap*]
eq-reflection[*OF Ex1-rev-unwrap*]
eq-reflection[*OF Ball-rev-unwrap*]
eq-reflection[*OF Bex-rev-unwrap*]

1.2.3 Cons reversion

definition *snocs* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where *snocs xs ys* = *xs* @ *ys*

lemma *Cons-rev*: *a # rev u = rev (snocs u [a])*
unfolding *snocs-def* **by** *simp*

lemma *snocs-snocs*: *snocs (snocs xs (y # ys)) zs = snocs xs (y # snocs ys zs)*
unfolding *snocs-def* **by** *simp*

1.2.4 Final corrections

lemma *snocs-Nil*: *snocs [] xs = xs*
unfolding *snocs-def* **by** *simp*

lemma *snocs-is-append*: *snocs xs ys = xs @ ys*
unfolding *snocs-def*..

private lemmas *final-correct1 =*
snocs-Nil

private lemmas *final-correct2 =*
snocs-is-append

1.2.5 Declaration attribute *reversal-rule*

ML ‹
structure Reversal-Rules =
Named-Thms(
val name = @{binding reversal-rule}
val description = Rules performing reverse−symmetry transformation on theorems on lists
)
›

attribute-setup *reversal-rule =*
‹*Attrib.add-del*
(*Thm.declaration-attribute Reversal-Rules.add-thm*)
(*Thm.declaration-attribute Reversal-Rules.del-thm*)›
maintaining a list of reversal rules

1.2.6 Rule attribute *reversed*

ML ‹
val eq-refl = @{thm eq-reflection}

fun pure-eq-of th =
case Thm.prop-of th of
*Const (**const-name**‹Pure.eq›, -) $ - $ -*
=> SOME (th)
*| Const (**const-name**‹Trueprop›, -) $ (Const (**const-name**‹HOL.eq›, -) $ - $ -*
)
=> SOME (th RS eq-refl)

```
  | - => NONE

val init-rev-wrap = @{thms init-rev-wrap}
val init-unwrap = @{thms init-rev-unwrap}
val final-correct1 = map-filter pure-eq-of @{thms final-correct1}
val final-correct2 = map-filter pure-eq-of @{thms final-correct2}

fun reverse ths context th =
  let
    val ctxt = Context.proof-of context
    val rules = map-filter pure-eq-of (ths @ Reversal-Rules.get ctxt)
    val vars = Term.add-vars (Thm.full-prop-of th) []
    fun add-inst-vars [] inst-vars = inst-vars
      | add-inst-vars ( ((v, i), T) :: vars ) inst-vars =
          case T of
            Type(type-name ‹list›, -) =>
              add-inst-vars vars (
                (
                  ((v, i), T),
                  Thm.cterm-of ctxt
                      ( Const(const-name ‹rev›, Type(type-name ‹fun›, [T, T])) $
Var((v, i), T) )
                ) :: inst-vars
              )
          | - => add-inst-vars vars inst-vars
  in
    th
    |> Drule.instantiate-normalize
       (TVars.empty, Vars.make (add-inst-vars vars []))
    |> Simplifier.rewrite-rule ctxt init-rev-wrap
    |> Simplifier.rewrite-rule ctxt init-unwrap
    |> Simplifier.rewrite-rule ctxt rules
    |> Simplifier.rewrite-rule ctxt final-correct1
    |> Simplifier.rewrite-rule ctxt final-correct2
  end

val reversed = Scan.optional (Scan.lift (Args.add -- Args.colon) |-- Attrib.thms)
[]
  >> (fn ths => Thm.rule-attribute [] (reverse ths))
›

attribute-setup reversed =
  ‹reversed›
  Transforms the theorem on lists to reverse−symmetric version

end
```

1.3 Declaration of basic reversal rules

lemma *hd-last-Nil*: *hd* [] = *last* []
unfolding *hd-def last-def* **by** *simp*

lemma *last-rev-hd*: *last*(*rev xs*) = *hd xs*
by (*induct xs*, *simp add*: *hd-last-Nil*, *simp*)

lemma *hd-rev-last*: *hd*(*rev xs*) = *last xs*
by (*induct xs*, *simp add*: *hd-last-Nil*, *simp*)

lemma *tl-rev*: *tl* (*rev xs*) = *rev* (*butlast xs*)
unfolding *rev-swap*[*symmetric*] *butlast-rev*[*of rev xs*, *symmetric*] *rev-rev-ident*..

lemma *if-rev*: (*if P then rev u else rev v*) = *rev* (*if P then u else v*)
by *simp*

lemma *rev-in-conv*: *rev u* ∈ *A* ⟷ *u* ∈ *rev* ‘ *A*
using *image-iff* **by** *fastforce*

lemma *in-lists-rev*: *u* ∈ *lists A* ⟹ *rev u* ∈ *lists A*
by *auto*

lemma *rev-in-lists*: *rev u* ∈ *lists A* ⟹ *u* ∈ *lists A*
by *auto*

lemma *rev-lists-conv*: *rev* ‘ *lists A* = *lists A*
proof (*intro equalityI subsetI*)
fix *x*
show *x* ∈ *rev* ‘ *lists A* ⟹ *x* ∈ *lists A*
unfolding *rev-in-conv*[*symmetric*] **using** *rev-in-lists*.
show *x* ∈ *lists A* ⟹ *x* ∈ *rev* ‘ *lists A*
unfolding *rev-in-conv*[*symmetric*] **using** *in-lists-rev*.
qed

lemma *coset-rev*: *List.coset* (*rev xs*) = *List.coset xs*
by *simp*

lemmas [*reversal-rule*] =
Cons-rev
snocs-snocs
rev-append[*symmetric*]
last-rev-hd hd-rev-last
tl-rev butlast-rev
rev-is-rev-conv
length-rev
take-rev
drop-rev
rotate-rev

if-rev
rev-in-conv
rev-lists-conv
set-rev
coset-rev

1.4 Examples

context
begin

1.4.1 Cons and append

private lemma *example-Cons-append*:
assumes $xs = [a, b]$ **and** $ys = [b, a, b]$
shows $xs \mathbin{@} xs \mathbin{@} xs = a \# b \# a \# ys$
using *assms* **by** *simp*

thm
example-Cons-append
example-Cons-append[*reversed*]
example-Cons-append[*reversed*, *reversed*]

thm
not-Cons-self
not-Cons-self[*reversed*]

thm
neq-Nil-conv
neq-Nil-conv[*reversed*]

1.4.2 Induction rules

thm
list-nonempty-induct
list-nonempty-induct[*reversed*]
list-nonempty-induct[*reversed*, **where** $P{=}\lambda x.\ P\ (rev\ x)$ **for** P, *unfolded rev-rev-ident*]

thm
list-induct2
list-induct2[*reversed*]
list-induct2[*reversed*, **where** $P{=}\lambda x\ y.\ P\ (rev\ x)\ (rev\ y)$ **for** P, *unfolded rev-rev-ident*]

1.4.3 hd, tl, last, butlast

thm
hd-append
hd-append[*reversed*]

last-append

thm
length-tl
length-tl[*reversed*]
length-butlast

thm
hd-Cons-tl
hd-Cons-tl[*reversed*]
append-butlast-last-id
append-butlast-last-id[*reversed*]

1.4.4 set

thm
hd-in-set
hd-in-set[*reversed*]
last-in-set

thm
set-ConsD
set-ConsD[*reversed*]

thm
split-list-first
split-list-first[*reversed*]

thm
split-list-first-prop
split-list-first-prop[*reversed*]
split-list-first-prop[*reversed*, *unfolded append-assoc append-Cons append-Nil*]
split-list-last-prop

thm
split-list-first-propE
split-list-first-propE[*reversed*]
split-list-first-propE[*reversed*, *unfolded append-assoc append-Cons append-Nil*]
split-list-last-propE

1.4.5 rotate

private lemma *rotate1-hd-tl'*: $xs \neq [] \Longrightarrow$ *rotate 1 xs* = *tl xs* @ [*hd xs*]
by (*cases xs*) *simp-all*

thm
rotate1-hd-tl'
rotate1-hd-tl'[*reversed*]

end

end

theory *CoWBasic*
  imports *HOL−Library.Sublist Arithmetical-Hints Reverse-Symmetry*
begin

Chapter 2

Basics of Combinatorics on Words

Combinatorics on Words, as the name suggests, studies words, finite or infinite sequences of elements from a, usually finite, alphabet. The essential operation on finite words is the concatenation of two words, which is associative and noncommutative. This operation yields many simply formulated problems, often in terms of *equations on words*, that are mathematically challenging.

See for instance [1] for an introduction to Combinatorics on Words, and [**?**, 5, 6] as another reference for Combinatorics on Words. This theory deals exclusively with finite words and provides basic facts of the field which can be considered as folklore.

The most natural way to represent finite words is by the type *'a list*. From an algebraic viewpoint, lists are free monoids. On the other hand, any free monoid is isomoporphic to a monoid of lists of its generators. The algebraic point of view and the combinatorial point of view therefore overlap significantly in Combinatorics on Words.

2.1 Definitions and notations

First, we introduce elementary definitions and notations.

The concatenation (@) of two finite lists/words is the very basic operation in Combinatorics on Words, its notation is usually omitted. In this field, a common notation for this operation is ·, which we use and add here.

notation *append* (**infixr** · *65*)

lemmas *rassoc* = *append-assoc*
lemmas *lassoc* = *append-assoc*[*symmetric*]

We add a common notation for the length of a given word $|w|$.

notation
length (|-|) — note that it's bold |
notation (*latex* **output**)
length (|-|)

2.1.1 Empty and nonempty word

As the word of length zero [] or [] will be used often, we adopt its frequent notation ε in this formalization.

notation *Nil* (ε)

abbreviation *drop-emp* :: *'a list set* ⇒ *'a list set* (-$_+$ [*1000*]) **where**
drop-emp S ≡ *S* − {ε}

2.1.2 Prefix

The property of being a prefix shall be frequently used, and we give it yet another frequent shorthand notation. Analogously, we introduce shorthand notations for non-empty prefix and strict prefix, and continue with suffixes and factors.

notation *prefix* (**infixl** ≤*p* *50*)
notation (*latex* **output**) *prefix* (≤$_p$)
lemmas [*simp*] = *prefix-def*

lemmas *prefI'*[*intro*] = *prefixI*

lemma *prefI*[*intro*]: *p* · *s* = *w* ⟹ *p* ≤*p* *w*
by *auto*

lemma *prefD*: *u* ≤*p* *v* ⟹ ∃ *z*. *v* = *u* · *z*
unfolding *prefix-def***.**

definition *prefix-comparable* :: *'a list* ⇒ *'a list* ⇒ *bool* (**infixl** ⋈ *50*)
where *prefix-comparable-def*[*simp*]: (*prefix-comparable u v*) ≡ *u* ≤*p* *v* ∨ *v* ≤*p* *u*

lemma *pref-compIff*[*iff*]: *u* ⋈ *v* ⟷ *u* ≤*p* *v* ∨ *v* ≤*p* *u*
by *simp*

definition *nonempty-prefix* (**infixl** ≤*np* *50*) **where** *nonempty-prefix-def*[*simp*]: *u* ≤*np* *v* ≡ *u* ≠ ε ∧ *u* ≤*p* *v*
notation (*latex* **output**) *nonempty-prefix* (≤$_{np}$ *50*)

lemma *npI*[*intro*]: *u* ≠ ε ⟹ *u* ≤*p* *v* ⟹ *u* ≤*np* *v*
by *auto*

lemma *npI'*[*intro*]: *u* ≠ ε ⟹ (∃ *z*. *u* · *z* = *v*) ⟹ *u* ≤*np* *v*

by *auto*

lemma *npD*: $u \leq np\ v \Longrightarrow u \leq p\ v$
by *simp*

lemma *npD'*: $u \leq np\ v \Longrightarrow u \neq \varepsilon$
by *simp*

notation *strict-prefix* (**infixl** $<p$ *50*)
notation (*latex* **output**) *strict-prefix* ($<_p$)
lemmas [*simp*] = *strict-prefix-def*

lemma *sprefI1*[*intro*]: $v = u \cdot z \Longrightarrow z \neq \varepsilon \Longrightarrow u <p\ v$
by *simp*

lemma *sprefI1'*[*intro*]: $u \cdot z = v \Longrightarrow z \neq \varepsilon \Longrightarrow u <p\ v$
by *blast*

lemma *sprefI2*[*intro*]: $u \leq p\ v \Longrightarrow length\ u < length\ v \Longrightarrow u <p\ v$
unfolding *strict-prefix-def*
by *blast*

lemma *sprefD*[*elim*]: $u <p\ v \Longrightarrow u \leq p\ v \wedge u \neq v$
by *auto*

lemmas *sprefD1*[*elim*] = *prefix-order.strict-implies-order* **and**
sprefD2[*elim*] = *prefix-order.less-imp-neq*

lemma *sprefE* [*elim*]: **assumes** $u <p\ v$ **obtains** z **where** $u \cdot z = v$ **and** $z \neq \varepsilon$
using *assms* **by** *auto*

2.1.3 Suffix

notation *suffix* (**infixl** $\leq s$ *60*)
notation (*latex* **output**) *suffix* ($\leq_s$)
lemmas [*simp*] = *suffix-def*

lemma *sufI*[*intro*]: $p \cdot s = w \Longrightarrow s \leq s\ w$
by *auto*

lemma *sufD*[*elim*]: $u \leq s\ v \Longrightarrow \exists\ z.\ z \cdot u = v$
by *auto*

notation *strict-suffix* (**infixl** $<s$ *50*)
notation (*latex* **output**) *strict-suffix* ($<_s$)
lemmas [*simp*] = *strict-suffix-def*

lemmas [*intro*] = *suffix-order.le-neq-trans*

lemma *ssufI1*[*intro*]: $u \cdot v = w \Longrightarrow u \neq \varepsilon \Longrightarrow v <_s w$
by *blast*

lemma *ssufI2*[*intro*]: $u \leq_s v \Longrightarrow$ *length* $u <$ *length* $v \Longrightarrow u <_s v$
by *blast*

lemma *ssufD*[*elim*]: $u <_s v \Longrightarrow u \leq_s v \wedge u \neq v$
by *auto*

lemmas *ssufD1*[*elim*] = *suffix-order.strict-implies-order* **and**
ssufD2[*elim*] = *suffix-order.less-imp-neq*

definition *suffix-comparable* :: *'a list* ⇒ *'a list* ⇒ *bool* (**infixl** $\bowtie_s$ *50*)
where *suffix-comparable-def*[*simp*]: (*suffix-comparable u v*) ≡ (*rev u*) ⋈ (*rev v*)

definition *nonempty-suffix* (**infixl** $\leq_{ns}$ *60*) **where** *nonempty-suffix-def*[*simp*]: $u \leq_{ns} v \equiv u \neq \varepsilon \wedge u \leq_s v$
notation (*latex* **output**) *nonempty-suffix* ($\leq_{ns}$ *50*)

lemma *nsI*[*intro*]: $u \neq \varepsilon \Longrightarrow u \leq_s v \Longrightarrow u \leq_{ns} v$
by *auto*

lemma *nsI'*[*intro*]: $u \neq \varepsilon \Longrightarrow (\exists\ z.\ z \cdot u = v) \Longrightarrow u \leq_{ns} v$
by *blast*

lemma *nsD*: $u \leq_{ns} v \Longrightarrow u \leq_s v$
by *simp*

lemma *nsD'*: $u \leq_{ns} v \Longrightarrow u \neq \varepsilon$
by *simp*

2.1.4 Factor

A *sublist* of some word is in Combinatorics of Words called a factor. We adopt a common shorthand notation for the property of being a factor, strict factor and nonempty factor (the latter we also define).

notation *sublist* (**infixl** $\leq_f$ *60*)
notation (*latex* **output**) *sublist* ($\leq_f$)
lemmas *factor-def*[*simp*] = *sublist-def*

notation *strict-sublist* (**infixl** $<_f$ *60*)
notation (*latex* **output**) *strict-sublist* ($<_f$)
lemmas *strict-factor-def*[*simp*] = *strict-sublist-def*

definition *nonempty-factor* (**infixl** $\leq_{nf}$ *60*) **where** *nonempty-factor-def*[*simp*]: $u \leq_{nf} v \equiv u \neq \varepsilon \wedge (\exists\ p\ s.\ p \cdot u \cdot s = v)$
notation (*latex* **output**) *nonempty-factor* ($\leq_{nf}$)

lemma *facI*: $u \leq_f p \cdot u \cdot s$
using *sublist-appendI*.

lemma *facI'*: $a \cdot u \cdot b = w \Longrightarrow u \leq_f w$
by *auto*

lemma *facE*[*elim*]: **assumes** $u \leq_f v$
obtains $p\ s$ **where** $v = p \cdot u \cdot s$
using *assms* **unfolding** *factor-def*
by *blast*

lemma *facE'*[*elim*]: **assumes** $u \leq_f v$
obtains $p\ s$ **where** $p \cdot u \cdot s = v$
using *assms* **unfolding** *factor-def*
by *blast*

2.2 Various elementary lemmas

lemmas *concat-morph* = *sym*[*OF concat-append*]

lemmas *cancel* = *same-append-eq* **and**
cancel-right = *append-same-eq*

lemma *bij-lists*: *bij-betw f X Y* $\Longrightarrow$ *bij-betw* (*map f*) (*lists X*) (*lists Y*)
proof−
assume *bij-betw f X Y*
hence *inj-on f X*
by (*simp add*: *bij-betw-def*)
have $\bigwedge$ *x y*. *x* $\in$ *lists X* $\Longrightarrow$ *y* $\in$ *lists X* $\Longrightarrow$ (*set x* $\cup$ *set y*) $\subseteq$ *X*
by *blast*
hence $\bigwedge$ *x y*. *x* $\in$ *lists X* $\Longrightarrow$ *y* $\in$ *lists X* $\Longrightarrow$ *inj-on f* (*set x* $\cup$ *set y*)
using *subset-inj-on*[*OF* ‹*inj-on f X*›] **by** *meson*
hence $\bigwedge$ *x y*. *x* $\in$ *lists X* $\Longrightarrow$ *y* $\in$ *lists X* $\Longrightarrow$ *map f x* = *map f y* $\longleftrightarrow$ *x* = *y*
by (*simp add*: *inj-on-map-eq-map*)
hence *inj-on* (*map f*) (*lists X*)
by (*simp add*: *inj-on-def*)
thus *?thesis* **using** ‹*bij-betw f X Y*› *bij-betw-def lists-image*
by *metis*
qed

lemma *concat-sing'*: *concat* [*r*] = *r*
by *simp*

lemma *concat-sing*: *s* = [*hd s*] $\Longrightarrow$ *concat s* = *hd s*
using *concat-sing'*[*of hd s*] **by** *auto*

lemma *rev-sing*: *rev* [*x*] = [*x*]
by *simp*

lemma *hd-word*: $a\#ws = [a] \cdot ws$
by *simp*

lemma *hd-word'*: $w \neq \varepsilon \Longrightarrow [hd\ w] \cdot tl\ w = w$
by *simp*

lemma *hd-pref*: $w \neq \varepsilon \Longrightarrow [hd\ w] \leq p\ w$
using *hd-word'*
by *blast*

lemma *add-nth*: **assumes** $n < |w|$ **shows** $(take\ n\ w) \cdot [w!n] \leq p\ w$
using *take-is-prefix*[*of Suc n w, unfolded take-Suc-conv-app-nth*[*OF assms*]]**.**

lemma *hd-pref'*: $w \neq \varepsilon \Longrightarrow [w!0] \leq p\ w$
using *add-nth* **by** *fastforce*

lemma *sub-lists-mono*: $A \subseteq B \Longrightarrow x \in lists\ A \Longrightarrow x \in lists\ B$
by *auto*

lemma *lists-hd*: $us \neq \varepsilon \Longrightarrow us \in lists\ Q \Longrightarrow hd\ us \in Q$
by *fastforce*

lemma *replicate-in-lists*: $replicate\ k\ z \in lists\ \{z\}$
by (*induction k*) *auto*

lemma *tl-lists*: **assumes** $us \in lists\ A$ **shows** $tl\ us \in lists\ A$
using *suffix-lists*[*OF suffix-tl assms*]**.**

lemma *long-list-tl*: **assumes** $1 < |us|$ **shows** $tl\ us \neq \varepsilon$
proof
assume $tl\ us = \varepsilon$
from *assms*
have $us \neq \varepsilon$ **and** $|us| = Suc\ |tl\ us|$ **and** $|us| \neq Suc\ 0$
by *auto*
thus *False*
using ‹$tl\ us = \varepsilon$› **by** *simp*
qed

lemma *tl-set*: $x \in set\ (tl\ a) \Longrightarrow x \in set\ a$
using *list.sel*(*2*) *list.set-sel*(*2*) **by** *metis*

lemma *drop-take-inv*: $n \leq |u| \Longrightarrow drop\ n\ (take\ n\ u \cdot w) = w$
by *simp*

lemma *split-list-long*: **assumes** $1 < |us|$ **and** $u \in set\ us$
obtains *xs ys* **where** $us = xs \cdot [u] \cdot ys$ **and** $xs \cdot ys \neq \varepsilon$
proof–
obtain *xs ys* **where** $us = xs \cdot [u] \cdot ys$

using *split-list-first*[*OF* ‹*u* ∈ *set us*›] **by** *auto*
hence *xs*·*ys*≠ε
using ‹*1* < |*us*|› **by** *auto*
from *that*[*OF* ‹*us* = *xs* · [*u*] · *ys*› *this*]
show *thesis*.
qed

lemma *flatten-lists*: *G* ⊆ *lists B* ⟹ *xs* ∈ *lists G* ⟹ *concat xs* ∈ *lists B*
proof (*induct xs*, *simp*)
case (*Cons a xs*)
hence *concat xs* ∈ *lists B* **and** *a* ∈ *lists B*
by *auto*
thus *?case*
by *simp*
qed

lemma *concat-map-sing-ident*: *concat* (*map* (λ *x*. [*x*]) *xs*) = *xs*
by *auto*

lemma *hd-concat-tl*: **assumes** *ws* ≠ ε **shows** *hd ws* · *concat* (*tl ws*) = *concat ws*
using *concat.simps*(*2*)[*of hd ws tl ws*, *unfolded list.collapse*[*OF* ‹*ws* ≠ ε›], *symmetric*].

lemma *concat-butlast-last*: **assumes** *ws* ≠ ε **shows** *concat* (*butlast ws*) · *last ws* = *concat ws*
using *concat-morph*[*of butlast ws* [*last ws*], *unfolded concat-sing*′ *append-butlast-last-id*[*OF* ‹*ws* ≠ ε›]].

lemma *concat-last-suf*: *ws* ≠ ε ⟹ *last ws* ≤*s concat ws*
using *concat-butlast-last* **by** *blast*

lemma *concat-hd-pref*: *ws* ≠ ε ⟹ *hd ws* ≤*p concat ws*
using *hd-concat-tl* **by** *blast*

lemma *set-nemp-concat-nemp*: **assumes** *ws* ≠ ε **and** ε ∉ *set ws* **shows** *concat ws* ≠ ε
using ‹ε ∉ *set ws*› *last-in-set*[*OF* ‹*ws* ≠ ε›] *concat-butlast-last*[*OF* ‹*ws* ≠ ε›] **by** *fastforce*

lemmas *takedrop* = *append-take-drop-id*

lemma *comm-rev-iff*: *rev u* · *rev v* = *rev v* · *rev u* ⟷ *u* · *v* = *v* · *u*
unfolding *rev-append*[*symmetric*] *rev-is-rev-conv eq-ac*(*1*)[*of u* · *v*] **by** *blast*

lemma *rev-induct2*:
⟦ *P* [] [];
⋀*x xs*. *P* (*xs*·[*x*]) [];
⋀*y ys*. *P* [] (*ys*·[*y*]);
⋀*x xs y ys*. *P xs ys* ⟹ *P* (*xs*·[*x*]) (*ys*·[*y*]) ⟧

$\Longrightarrow$ *P xs ys*
proof (*induct xs arbitrary*: *ys rule*: *rev-induct*)
case *Nil*
then show *?case*
using *rev-induct*[*of P* ε]
by *presburger*
next
case (*snoc x xs*)
hence *P xs ys'* **for** *ys'*
by *simp*
then show *?case*
by (*simp add*: *rev-induct snoc.prems*(*2*) *snoc.prems*(*4*))
qed

2.2.1 Orderings on lists: prefix, suffix, factor

lemmas *self-pref* = *prefix-order.refl*
lemmas *pref-antisym* = *prefix-order.antisym*
lemmas *pref-trans* = *prefix-order.trans*
lemmas *suf-trans* = *suffix-order.trans*

2.2.2 On the empty word

lemma *nemp-elem-setI*[*intro*]: $u \in S \Longrightarrow u \neq \varepsilon \Longrightarrow u \in S_+$
by *simp*

lemma *nel-drop-emp*: $u \neq \varepsilon \Longrightarrow u \in S \Longrightarrow u \in S_+$
by *simp*

lemma *drop-emp-nel*: **assumes** $u \in S_+$ **shows** $u \neq \varepsilon$ **and** $u \in S$
using *assms* **by** *simp*+

lemma *emp-concat-emp*: *us* $\in$ *lists* $S_+ \Longrightarrow$ *concat us* $= \varepsilon \Longrightarrow$ *us* $= \varepsilon$
using *DiffD2* **by** *auto*

lemma *take-nemp*: $w \neq \varepsilon \Longrightarrow n \neq 0 \Longrightarrow$ *take n w* $\neq \varepsilon$
by *simp*

lemma *pref-nemp* [*intro*]: $u \neq \varepsilon \Longrightarrow u \cdot v \neq \varepsilon$
unfolding *append-is-Nil-conv* **by** *simp*

lemma *suf-nemp* [*intro*]: $v \neq \varepsilon \Longrightarrow u \cdot v \neq \varepsilon$
unfolding *append-is-Nil-conv* **by** *simp*

2.3 Length and its properties

lemma *lenarg*: $u = v \Longrightarrow |u| = |v|$
by *simp*

lemma *npos-len*: $|u| \leq 0 \Longrightarrow u = \varepsilon$
by *simp*

lemma *nemp-pos-len*: $r \neq \varepsilon \Longrightarrow 1 \leq |r|$
by (*simp add*: *leI*)

lemma *swap-len*: $|u \cdot v| = |v \cdot u|$
by *simp*

lemma *len-after-drop*: $p + q \leq |w| \Longrightarrow q \leq |drop\ p\ w|$
by *simp*

lemma *short-take-append*: $n \leq |u| \Longrightarrow take\ n\ (u \cdot v) = take\ n\ u$
by *simp*

lemma *sing-word*: $|us| = 1 \Longrightarrow [hd\ us] = us$
by (*cases us*) *simp+*

lemma *sing-word-concat*: **assumes** $|us| = 1$ **shows** $[concat\ us] = us$
by (*simp add*: *assms concat-sing sing-word*)

lemma *nonsing-concat-len*: $|us| \neq 1 \Longrightarrow concat\ us \neq \varepsilon \Longrightarrow 1 < |us|$
using *nat-neq-iff* **by** *fastforce*

lemma *sing-len*: $|[a]| = 1$
by *simp*

lemma *pref-len'*: $|u| \leq |u \cdot z|$
by *auto*

lemma *suf-len'*: $|u| \leq |z \cdot u|$
by *auto*

lemma *fac-len*: $u \leq f\ v \Longrightarrow |u| \leq |v|$
by *auto*

lemma *fac-len'*: $|w| \leq |u \cdot w \cdot v|$
by *simp*

lemma *fac-len-eq*: $u \leq f\ v \Longrightarrow |u| = |v| \Longrightarrow u = v$
unfolding *factor-def* **using** *length-append npos-len* **by** *fastforce*

lemma *drop-len*: $|u \cdot w| \leq |u \cdot v \cdot w|$
by *simp*

lemma *drop-pref*: $drop\ |u|\ (u \cdot w) = w$
by *simp*

lemma *take-len*: $p \leq |w| \Longrightarrow |take\ p\ w| = p$

using *trans*[*OF length-take min-absorb2*].

lemma *conj-len*: $p \cdot x = x \cdot s \Longrightarrow |p| = |s|$
using *length-append*[*of p x*] *length-append*[*of x s*] *add.commute add-left-imp-eq*
by *auto*

lemma *take-nemp-len*: $u \neq \varepsilon \Longrightarrow r \neq \varepsilon \Longrightarrow take\ |r|\ u \neq \varepsilon$
by *simp*

lemma *nemp-len*: $u \neq \varepsilon \Longrightarrow |u| \neq 0$
by *simp*

lemma *take-self*: $take\ |w|\ w = w$
using *take-all*[*of w* $|w|$, *OF order.refl*].

lemma *len-le-concat*: $\varepsilon \notin set\ ws \Longrightarrow |ws| \leq |concat\ ws|$
proof (*induct ws*, *simp*)
case (*Cons a ws*)
hence $1 \leq |a|$
using *list.set-intros*(*1*)[*of a ws*] *nemp-pos-len*[*of a*] **by** *blast*
then show *?case*
unfolding *concat.simps*(*2*) **unfolding** *length-append hd-word*[*of a ws*] *sing-len*
using *Cons.hyps Cons.prems* **by** *simp*
qed

lemma *eq-len-iff*: **assumes** *eq*: $x \cdot y = u \cdot v$ **shows** $|x| \leq |u| \longleftrightarrow |v| \leq |y|$
using *lenarg*[*OF eq*] **unfolding** *length-append* **by** *auto*

lemma *eq-len-iff-less*: **assumes** *eq*: $x \cdot y = u \cdot v$ **shows** $|x| < |u| \longleftrightarrow |v| < |y|$
using *lenarg*[*OF eq*] **unfolding** *length-append* **by** *auto*

2.4 Prefix and prefix comparability properties

lemmas *pref-emp* = *prefix-bot.extremum-uniqueI*

lemma *triv-pref*: $r \leq p\ r \cdot s$
using *prefI*[*OF refl*].

lemma *triv-spref*: $s \neq \varepsilon \Longrightarrow r <p\ r \cdot s$
by *simp*

lemma *pref-cancel*: $z \cdot u \leq p\ z \cdot v \Longrightarrow u \leq p\ v$
by *simp*

lemma *pref-cancel'*: $u \leq p\ v \Longrightarrow z \cdot u \leq p\ z \cdot v$
by *simp*

lemmas *pref-cancel-conv* = *same-prefix-prefix*

lemmas *pref-ext* = *prefix-prefix* — provided by Sublist.thy

lemma *spref-ext*: $r <p\ u \Longrightarrow r <p\ u \cdot v$
by *force*

lemma *pref-ext-nemp*: $r \leq p\ u \Longrightarrow v \neq \varepsilon \Longrightarrow r <p\ u \cdot v$
by *auto*

lemma *pref-take*: $p \leq p\ w \Longrightarrow take\ |p|\ w = p$
by *auto*

lemma *pref-take-conv*: $take\ (|r|)\ w = r \longleftrightarrow r \leq p\ w$
using *pref-take*[*of r w*] *take-is-prefix*[*of* $|r|$ *w*] **by** *argo*

lemma *le-suf-drop*: **assumes** $i \leq j$ **shows** *drop j w* $\leq s$ *drop i w*
using *suffix-drop*[*of* $j - i$ *drop i w*, *unfolded drop-drop le-add-diff-inverse2*[*OF* ‹$i \leq j$›]]**.**

lemma *spref-take*: $p <p\ w \Longrightarrow take\ |p|\ w = p$
by *auto*

lemma *pref-same-len*: $u \leq p\ v \Longrightarrow |u| = |v| \Longrightarrow u = v$
by *auto*

lemma *add-nth-pref*: **assumes** $u <p\ w$ **shows** $u \cdot [w!|u|] \leq p\ w$
using *add-nth*[*OF prefix-length-less*[*OF* ‹$u <p\ w$›], *unfolded spref-take*[*OF* ‹$u <p\ w$›]]**.**

lemma *index-pref*: $|u| \leq |w| \Longrightarrow (\forall\ i < |u|.\ \ u!i = w!i) \Longrightarrow u \leq p\ w$
using *trans*[*OF sym*[*OF take-all*[*OF order-refl*]] *nth-take-lemma*[*OF order-refl*], *of u w*]
take-is-prefix[*of* $|u|$ *w*] **by** *auto*

lemma *pref-index*: **assumes** $u \leq p\ w\ i < |u|$ **shows** $u!i = w!i$
using *nth-take*[*OF* ‹$i < |u|$›, *of w*, *unfolded pref-take*[*OF* ‹$u \leq p\ w$›]]**.**

lemma *pref-drop*: $u \leq p\ v \Longrightarrow drop\ p\ u \leq p\ drop\ p\ v$
using *prefI*[*OF sym*[*OF drop-append*]] **by** *auto*

2.4.1 Prefix comparability

lemma *pref-comp-sym*[*sym*]: $u \bowtie v \Longrightarrow v \bowtie u$
by *blast*

lemmas *ruler-le* = *prefix-length-prefix* **and**
ruler = *prefix-same-cases* **and**
ruler′ = *prefix-same-cases*[*folded prefix-comparable-def*]

lemma *ruler-equal*: $u \leq p\ w \Longrightarrow v \leq p\ w \Longrightarrow |u| = |v| \Longrightarrow u = v$

by *auto*

lemma *ruler-comp*: $u \le_p v \Longrightarrow u' \le_p v' \Longrightarrow v \bowtie v' \Longrightarrow u \bowtie u'$
unfolding *prefix-comparable-def*
using *disjE*[*OF - ruler*[*OF pref-trans*] *ruler*[*OF - pref-trans*]]**.**

lemma *ruler-pref*: $w \le_p v{\cdot}z \Longrightarrow w \bowtie v$
unfolding *prefix-comparable-def*
using *ruler* **by** *blast*

lemma *pref-prod-pref-short*: $u \le_p z \cdot w \Longrightarrow v \le_p w \Longrightarrow |u| \le |z \cdot v| \Longrightarrow u \le_p z \cdot v$
using *ruler-le*[*OF - pref-cancel'*]**.**

lemma *pref-prod-pref*: $u \le_p z \cdot w \Longrightarrow u \le_p w \Longrightarrow u \le_p z \cdot u$
using *pref-prod-pref-short*[*OF - - suf-len'*]**.**

lemma *pref-prod-pref'*: **assumes** $u \le_p z{\cdot}u{\cdot}w$ **shows** $u \le_p z{\cdot}u$
using *pref-prod-pref*[*of u z u · w*, *OF* ‹$u \le_p z{\cdot}u{\cdot}w$› *triv-pref*]**.**

lemma *pref-prod-long*: $u \le_p v \cdot w \Longrightarrow |v| \le |u| \Longrightarrow v \le_p u$
using *ruler-le*[*OF triv-pref*]**.**

lemma *pref-keeps-root*: $u \le_p r \cdot u \Longrightarrow v \le_p u \Longrightarrow v \le_p r \cdot v$
using *pref-prod-pref*[*of v r u*] *pref-trans*[*of v u r·u*] **by** *blast*

lemma *pref-prolong*: $w \le_p z \cdot r \Longrightarrow r \le_p s \Longrightarrow w \le_p z \cdot s$
using *pref-trans*[*OF - pref-cancel'*]**.**

lemma *pref-prolong'*: **assumes** $u \le_p w \cdot z\ v \cdot u \le_p z$ **shows** $u \le_p w \cdot v \cdot u$
using *prefix-length-prefix*[*OF* ‹$u \le_p w \cdot z$› *pref-cancel'*[*OF* ‹$v \cdot u \le_p z$›, *of w*] *suf-len'*[*of u w·v*, *unfolded rassoc*]]**.**

lemma *pref-prolong-comp*: $u \le_p w \cdot z \Longrightarrow v \cdot u \bowtie z \Longrightarrow u \le_p w \cdot v \cdot u$
using *pref-prolong*[*of u w z v · u*] *pref-prolong'*[*of u w z v*] **by** *blast*

lemma *pref-prod-short*: $u \le_p v \cdot w \Longrightarrow |u| \le |v| \Longrightarrow u \le_p v$
using *prefI prefix-length-prefix*[*of u v·w v*]
by *blast*

lemma *pref-prod-short'*: **assumes** $u \le_p v \cdot w$ **and** $|u| < |v|$ **shows** $u <_p v$
using *pref-prod-short*[*OF* ‹$u \le_p v \cdot w$› *less-imp-le*[*OF* ‹$|u| < |v|$›]] ‹$|u| < |v|$›
by *blast*

lemma *pref-prod-cancel*: **assumes** $u \le_p p{\cdot}w{\cdot}q$ **and** $|p| \le |u|$ **and** $|u| \le |p{\cdot}w|$
obtains r **where** $u = p \cdot r$ **and** $r \le_p w$
proof−
have $p \le_p u$
using *pref-prod-long*[*OF* ‹$u \le_p p{\cdot}w{\cdot}q$› ‹$|p| \le |u|$›]**.**

then obtain r **where** $u = p \cdot r$ **using** *prefD* **by** *blast*
hence $r \leq_p w$ **using** $\langle |u| \leq |p{\cdot}w| \rangle$ $\langle u \leq_p p{\cdot}w{\cdot}q \rangle$
unfolding $\langle u = p \cdot r \rangle$ *pref-cancel-conv length-append* **using** *pref-prod-short*[*of* r w q] **by** *simp*
from *that*[*OF* $\langle u = p \cdot r \rangle$ *this*]
show *thesis*.
qed

lemma *pref-prod-cancel'*: **assumes** $u \leq_p p{\cdot}w{\cdot}q$ **and** $|p| < |u|$ **and** $|u| \leq |p{\cdot}w|$
obtains r **where** $u = p \cdot r$ **and** $r \leq_p w$ **and** $r \neq \varepsilon$
proof−
obtain r **where** $u = p \cdot r$ **and** $r \leq_p w$
using *pref-prod-cancel*[*OF* $\langle u \leq_p p{\cdot}w{\cdot}q \rangle$ *less-imp-le*[*OF* $\langle |p| < |u| \rangle$] $\langle |u| \leq |p{\cdot}w| \rangle$].
moreover have $r \neq \varepsilon$ **using** $\langle u = p \cdot r \rangle$ *less-not-refl3*[*OF* $\langle |p| < |u| \rangle$, *folded self-append-conv*] **by** *simp*
ultimately show *thesis* **using** *that* **by** *simp*
qed

lemma *pref-comp-eq*: $u \bowtie v \Longrightarrow |u| = |v| \Longrightarrow u = v$
by *auto*

lemma *non-comp-parallel*: $\neg\ u \bowtie v \longleftrightarrow u \parallel v$
unfolding *prefix-comparable-def parallel-def de-Morgan-disj*..

lemma *comp-refl*: $u \bowtie u$
by *simp*

lemma *incomp-cancel*: $\neg\ p{\cdot}u \bowtie p{\cdot}v \Longrightarrow \neg\ u \bowtie v$
by *simp*

lemma *comp-cancel*: $z \cdot u \bowtie z \cdot v \Longrightarrow u \bowtie v$
by *simp*

lemma *comm-ruler*: $r \cdot s \leq_p w1 \Longrightarrow s \cdot r \leq_p w2 \Longrightarrow w1 \bowtie w2 \Longrightarrow r \cdot s = s \cdot r$
using *pref-comp-eq*[*OF ruler-comp swap-len*].

lemma *pref-share-take*: $u \leq_p v \Longrightarrow q \leq |u| \Longrightarrow take\ q\ u = take\ q\ v$
by *auto*

lemma *pref-prod-longer*: $u \leq_p z \cdot w \Longrightarrow v \leq_p w \Longrightarrow |z \cdot v| \leq |u| \Longrightarrow z \cdot v \leq_p u$
using *ruler-le*[*OF pref-cancel'*].

lemma *pref-comp-not-pref*: $u \bowtie v \Longrightarrow \neg\ v \leq_p u \Longrightarrow u <_p v$
by *auto*

lemma *pref-comp-not-spref*: $u \bowtie v \Longrightarrow \neg\ u <_p v \Longrightarrow v \leq_p u$
using *contrapos-np*[*OF - pref-comp-not-pref*].

lemma *hd-prod*: $u \neq \varepsilon \Longrightarrow (u \cdot v)!0 = u!0$
by (*cases u*) (*blast*, *simp*)

lemma *distinct-first*: **assumes** $w \neq \varepsilon$ $z \neq \varepsilon$ $w!0 \neq z!0$ **shows** $w \cdot w' \neq z \cdot z'$
using *hd-prod*[*of w w'*, *OF* ‹$w \neq \varepsilon$›] *hd-prod*[*of z z'*, *OF* ‹$z \neq \varepsilon$›] ‹$w!0 \neq z!0$› **by** *auto*

lemmas *last-no-split* = *prefix-snoc*

lemma *last-no-split'*: **assumes** $u <p\ w$ $w \leq p\ u \cdot [a]$ **shows** $w = u \cdot [a]$
using *assms*(*1*)[*unfolded prefix-order.less-le-not-le*] *assms*(*2*)[*unfolded last-no-split*]
by *presburger*

lemma *pcomp-shorter*: $v \bowtie w \Longrightarrow |v| \leq |w| \Longrightarrow v \leq p\ w$
by *auto*

lemma *pref-comp-len-trans*: $w \leq p\ v \Longrightarrow u \bowtie v \Longrightarrow |w| \leq |u| \Longrightarrow w \leq p\ u$
unfolding *prefix-comparable-def*
using *prefix-length-prefix*[*of w v u*] *prefix-order.trans*[*of w v u*]
by *argo*

lemma *comp-ext*: $z \cdot w1 \bowtie z \cdot w2 \longleftrightarrow w1 \bowtie w2$
using *pref-cancel* **by** *auto*

lemma *emp-pref*: $\varepsilon \leq p\ u$
by *simp*

lemma *emp-spref*: $u \neq \varepsilon \Longrightarrow \varepsilon <p\ u$
by *simp*

lemma *long-pref*: $u \leq p\ v \Longrightarrow |v| \leq |u| \Longrightarrow u = v$
by *auto*

lemma *incomp-ext*: $\neg\ w1 \bowtie\ w2 \Longrightarrow \neg\ w1 \cdot z \bowtie w2 \cdot z'$
using *contrapos-nn*[*OF - ruler-comp*[*OF triv-pref triv-pref*]]**.**

lemma *mismatch-incopm*: $|u| = |v| \Longrightarrow x \neq y \Longrightarrow \neg\ u \cdot [x] \bowtie v \cdot [y]$
by *simp*

lemma *comp-prefs-comp*: $u \cdot z \bowtie v \cdot w \Longrightarrow u \bowtie v$
using *ruler-comp*[*OF prefI*[*of - z*] *prefI*[*of - w*], *OF refl refl*]**.**

lemma *comp-hd-eq*: $u \bowtie v \Longrightarrow u \neq \varepsilon \Longrightarrow v \neq \varepsilon \Longrightarrow hd\ u = hd\ v$
by *auto*

lemma *pref-hd-eq'*: $p \leq p\ u \Longrightarrow p \leq p\ v \Longrightarrow p \neq \varepsilon \Longrightarrow\ hd\ u = hd\ v$
by *auto*

lemma *pref-hd-eq*: $u \leq p\ v \Longrightarrow u \neq \varepsilon \Longrightarrow\ hd\ u = hd\ v$
by *auto*

lemma *suf-last-eq*: $p \leq s\ u \Longrightarrow p \leq s\ v \Longrightarrow p \neq \varepsilon \Longrightarrow\ last\ u = last\ v$
by *auto*

lemma *comp-hd-eq'*: **assumes** $u \cdot r \bowtie v \cdot s\ u \neq \varepsilon\ v \neq \varepsilon$ **shows** $hd\ u = hd\ v$
using *comp-prefs-comp*[*OF* ‹$u \cdot r \bowtie v \cdot s$›] ‹$u \neq \varepsilon$› ‹$v \neq \varepsilon$› **by** *auto*

2.5 Suffix and suffix comparability properties

lemmas *suf-emp* = *suffix-bot.extremum-uniqueI*

lemma *triv-suf*: $u \leq s\ v \cdot u$
by *simp*

lemma *emp-ssuf*: $u \neq \varepsilon \Longrightarrow \varepsilon < s\ u$
by *simp*

lemma *suf-cancel*: $u \cdot v \leq s\ w \cdot v \Longrightarrow u \leq s\ w$
by *simp*

lemma *suf-cancel'*: $u \leq s\ w \Longrightarrow u \cdot v \leq s\ w \cdot v$
by *simp*

lemmas *suf-cancel-eq* = *same-suffix-suffix* — provided by Sublist.thy

Straightforward relations of suffix and prefix follow.

lemmas *suf-rev-pref-iff* = *suffix-to-prefix* — provided by Sublist.thy

lemmas *ssuf-rev-pref-iff* = *strict-suffix-to-prefix* — provided by Sublist.thy

lemma *pref-rev-suf-iff*: $u \leq p\ v \longleftrightarrow rev\ u \leq s\ rev\ v$
using *suffix-to-prefix*[*of rev u rev v*] **unfolding** *rev-rev-ident*
by *blast*

lemma *spref-rev-suf-iff*: $s < p\ w \longleftrightarrow rev\ s < s\ rev\ w$
using *strict-suffix-to-prefix*[*of rev s rev w*, *unfolded rev-rev-ident*, *symmetric*]**.**

lemma *nsuf-rev-pref-iff*: $s \leq ns\ w \longleftrightarrow rev\ s \leq np\ rev\ w$
unfolding *nonempty-prefix-def nonempty-suffix-def suffix-to-prefix*
by *fast*

lemma *npref-rev-suf-iff*: $s \leq np\ w \longleftrightarrow rev\ s \leq ns\ rev\ w$
unfolding *nonempty-prefix-def nonempty-suffix-def pref-rev-suf-iff*
by *fast*

lemmas [*reversal-rule*] =
suf-rev-pref-iff[*symmetric*]

pref-rev-suf-iff[*symmetric*]
nsuf-rev-pref-iff[*symmetric*]
npref-rev-suf-iff[*symmetric*]
ssuf-rev-pref-iff[*symmetric*]
spref-rev-suf-iff[*symmetric*]

lemmas *suf-ext* = *suffix-appendI* — provided by Sublist.thy

lemmas *ssuf-ext* = *spref-ext*[*reversed*] **and**
suf-ext-nem = *pref-ext-nemp*[*reversed*] **and**
suf-same-len = *pref-same-len*[*reversed*] **and**
suf-take = *pref-drop*[*reversed*] **and**
suf-share-take = *pref-share-take*[*reversed*] **and**
long-suf = *long-pref*[*reversed*]

2.5.1 Suffix comparability

lemma *pref-comp-rev-suf-comp*[*reversal-rule*]: $(rev\ w) \bowtie_s (rev\ v) \longleftrightarrow w \bowtie v$
by *simp*

lemma *suf-comp-rev-pref-comp*[*reversal-rule*]: $(rev\ w) \bowtie (rev\ v) \longleftrightarrow w \bowtie_s v$
by *simp*

lemmas *suf-ruler-le* = *suffix-length-suffix* — provided by Sublist.thy, same as ruler_le[reversed]

lemmas *suf-ruler* = *suffix-same-cases* — provided by Sublist.thy, same as ruler[reversed]

lemmas *suf-ruler-equal* = *ruler-equal*[*reversed*] **and**
suf-ruler-comp = *ruler-comp*[*reversed*] **and**
ruler-suf = *ruler-pref*[*reversed*] **and**
suf-prod-short = *pref-prod-short*[*reversed*] **and**
suf-prod-short′ = *pref-prod-short′*[*reversed*] **and**
suf-prod-cancel = *pref-prod-cancel*[*reversed*] **and**
suf-prod-cancel′ = *pref-prod-cancel′*[*reversed*] **and**
suf-prod-suf-short = *pref-prod-pref-short*[*reversed*] **and**
suf-prod-suf = *pref-prod-pref*[*reversed*] **and**
suf-prod-suf′ = *pref-prod-pref′*[*reversed*, *unfolded rassoc*] **and**
suf-prolong = *pref-prolong*[*reversed*] **and**
suf-prolong′ = *pref-prolong′*[*reversed*, *unfolded rassoc*] **and**
suf-prolong-comp = *pref-prolong-comp*[*reversed*, *unfolded rassoc*] **and**
suf-prod-long = *pref-prod-long*[*reversed*] **and**
suf-prod-longer = *pref-prod-longer*[*reversed*] **and**
suf-keeps-root = *pref-keeps-root*[*reversed*] **and**
comm-suf-ruler = *comm-ruler*[*reversed*]

lemmas *comp-sufs-comp* = *comp-prefs-comp*[*reversed*] **and**
suf-comp-not-suf = *pref-comp-not-pref*[*reversed*] **and**

suf-comp-not-ssuf = *pref-comp-not-spref*[*reversed*] **and**

suf-comp-ext = *comp-ext*[*reversed*] **and**
suf-incomp-ext = *incomp-ext*[*reversed*] **and**
mismatch-suf-incopm = *mismatch-incopm*[*reversed*] **and**
suf-comp-sym[*sym*] = *pref-comp-sym*[*reversed*]

lemma *suf-comp-last-eq*: **assumes** $u \bowtie_s v$ $u \neq \varepsilon$ $v \neq \varepsilon$
shows *last u* = *last v*
using *comp-hd-eq*[*reversed*, *OF assms*] **unfolding** *hd-rev hd-rev* **.**

lemma *suf-comp-last-eq'*: $r \cdot u \bowtie_s s \cdot v \Longrightarrow u \neq \varepsilon \Longrightarrow v \neq \varepsilon \Longrightarrow$ *last u* = *last v*
using *comp-sufs-comp suf-comp-last-eq* **by** *blast*

2.6 Left and Right Quotient

A useful function of left quotient is given. Note that the function is sometimes undefined.

definition *left-quotient*:: *'a list* ⇒ *'a list* ⇒ *'a list* ((-$^{-1>}$)(-) [*75*,*74*] *74*)
where *left-quotient-def*[*simp*]: *left-quotient u v* = (*if u* ≤*p v then* (*THE z*. *u* · *z* = *v*) *else undefined*)
notation (*latex* **output**) *left-quotient* (- $^{-1}$ · -)

Analogously, we define the right quotient.

definition *right-quotient* :: *'a list* ⇒ *'a list* ⇒ *'a list* ((-)($^{<-1}$-) [*76*,*77*] *76*)
where *right-quotient-def*[*simp*]: *right-quotient u v* = *rev* ((*rev v*)$^{-1>}$(*rev u*))
notation (*latex* **output**) *right-quotient* (- · - $^{-1}$)

Priorities of these operations are as follows:

lemma $u^{<-1}v^{<-1}w = (u^{<-1}v)^{<-1}w$
by *simp*

lemma $u^{-1>}v^{-1>}w = u^{-1>}(v^{-1>}w)$
by *simp*

lemma $u^{-1>}v^{<-1}w = u^{-1>}(v^{<-1}w)$
by *simp*

lemma $r \cdot u^{-1>}w^{<-1}v \cdot s = r \cdot (u^{-1>}w^{<-1}v) \cdot s$
by *simp*

lemma *rq-rev-lq*[*reversal-rule*]: (*rev v*)$^{<-1}$(*rev u*) = *rev* ($u^{-1>}v$)
by *simp*

lemma *lq-rev-rq*[*reversal-rule*]: (*rev v*)$^{-1>}$*rev u* = *rev* ($u^{<-1}v$)
by *simp*

2.6.1 Left Quotient

lemma *lqI*: $u \cdot z = v \Longrightarrow u^{-1>}v = z$
by *auto*

lemma *lq-triv*[*simp*]: $u^{-1>}(u \cdot z) = z$
using *lqI*[*OF refl*]**.**

lemma *lq-triv'*[*simp*]: $u \cdot u^{-1>}(u \cdot z) = u \cdot z$
by *simp*

lemma *lq-self*[*simp*]: $u^{-1>}u = \varepsilon$
by *auto*

lemma *lq-emp*[*simp*]: $\varepsilon^{-1>}u = u$
by *auto*

lemma *lq-pref*[*simp*]: $u \leq p\ v \Longrightarrow u \cdot (u^{-1>}v) = v$
by *auto*

lemma *lq-the*[*simp*]: $u \leq p\ v \Longrightarrow (u^{-1>}v) = (THE\ z.\ u \cdot z = v)$
by *simp*

lemma *lq-reassoc*: $u \leq p\ v \Longrightarrow (u^{-1>}v) \cdot w = u^{-1>}(v \cdot w)$
by *auto*

lemma *lq-trans*: $u \leq p\ v \Longrightarrow v \leq p\ w \Longrightarrow (u^{-1>}v) \cdot (v^{-1>}w) = u^{-1>}w$
by *auto*

lemma *lq-rq-reassoc-suf*: $u \leq p\ z \Longrightarrow u \leq s\ w \Longrightarrow w \cdot u^{-1>}z = w^{<-1}u \cdot z$
using *lq-pref*[*reversed*]
by *fastforce*

lemma *lq-ne*: $p \leq p\ u \cdot p \Longrightarrow u \neq \varepsilon \Longrightarrow p^{-1>}(u \cdot p) \neq \varepsilon$
using *lq-pref*[*of p u · p*] **by** *fastforce*

lemma *lq-spref*: $u <p\ v \Longrightarrow u^{-1>}v \neq \varepsilon$
using *lq-pref* **by** *auto*

lemma *lq-suf-suf*: $r \leq p\ s \Longrightarrow (r^{-1>}s) \leq s\ s$
by *auto*

lemma *lq-len*: $r \leq p\ s \Longrightarrow |r| + |r^{-1>}s| = |s|$
by *auto*

lemma *pref-lq*: $u \leq p\ v \Longrightarrow v \leq p\ w \Longrightarrow u^{-1>}v \leq p\ u^{-1>}w$
by *auto*

lemma *spref-lq*: $u \leq p\ v \Longrightarrow v <p\ w \Longrightarrow u^{-1>}v <p\ u^{-1>}w$
by *force*

lemma *conjug-lq*: $x \cdot z = z \cdot y \Longrightarrow y = z^{-1>}(x \cdot z)$
by *simp*

lemma *conjug-emp-emp*: $p \leq p\ u \cdot p \Longrightarrow p^{-1>}(u \cdot p) = \varepsilon \Longrightarrow u = \varepsilon$
using *lq-ne* **by** *blast*

lemma *lq-drop*: $u \leq p\ v \Longrightarrow u^{-1>}v = drop\ |u|\ v$
by *fastforce*

lemma *lq-code* [*code*]:
left-quotient $\varepsilon\ v = v$
left-quotient $(a\#u)\ \varepsilon = undefined$
left-quotient $(a\#u)\ (b\#v) = (if\ a{=}b\ then\ left\text{-}quotient\ u\ v\ else\ undefined)$
by *simp-all*

2.6.2 Right quotient

lemmas *rqI* = *lqI*[*reversed*] **and**
rq-triv = *lq-triv*[*reversed*] **and**
rq-triv' = *lq-triv'*[*reversed*] **and**
rq-seflt = *lq-self*[*reversed*] **and**
rq-emp = *lq-emp*[*reversed*] **and**
rq-suf = *lq-pref*[*reversed*] **and**
rq-ssuf = *lq-spref*[*reversed*] **and**
rq-reassoc = *lq-reassoc*[*reversed*] **and**
rq-len = *lq-len*[*reversed*] **and**
rq-trans = *lq-trans*[*reversed*] **and**
rq-lq-reassoc-suf = *lq-rq-reassoc-suf*[*reversed*] **and**
rq-ne = *lq-ne*[*reversed*] **and**
rq-suf-suf = *lq-suf-suf*[*reversed*] **and**
suf-rq = *pref-lq*[*reversed*] **and**
ssuf-rq = *spref-lq*[*reversed*] **and**
conjug-rq = *conjug-lq*[*reversed*] **and**
conjug-emp-emp' = *conjug-emp-emp*[*reversed*] **and**
rq-take = *lq-drop*[*reversed*]

2.6.3 Left and right quotients combined

lemma *rev-lq'*: $r \leq p\ s \Longrightarrow rev\ (r^{-1>}s) = (rev\ s)^{<-1}(rev\ r)$
by *auto*

lemma *pref-rq-suf-lq*: $s \leq s\ u \Longrightarrow r \leq p\ (u^{<-1}s) \Longrightarrow s \leq s\ (r^{-1>}u)$
using *lq-reassoc*[*of r* $u^{<-1}s$ *s*] *rq-suf*[*of s u*] *triv-suf*[*of s* $r^{-1>}u^{<-1}s$]
by *presburger*

lemmas *suf-lq-pref-rq* = *pref-rq-suf-lq*[*reversed*]

lemma $w{\cdot}s = v \Longrightarrow v^{<-1}s = w$ **using** *rqI*.

lemma *lq-rq-assoc*: $s \leq s\ u \Longrightarrow r \leq p\ (u^{<-1}s) \Longrightarrow (r^{-1>}u)^{<-1}s = r^{-1>}(u^{<-1}s)$
using *lq-reassoc*[*of r* $u^{<-1}s$ *s*] *rq-suf*[*of s u*] *rqI*[*of* $r^{-1>}u^{<-1}s$ *s* $r^{-1>}u$]
by *argo*

lemmas *rq-lq-assoc* = *lq-rq-assoc*[*reversed*]

lemma *lq-prod*: $u \leq p\ v{\cdot}u \Longrightarrow u \leq p\ w \Longrightarrow\ u^{-1>}(v{\cdot}u){\cdot}u^{-1>}w = u^{-1>}(v{\cdot}w)$
using *lq-reassoc*[*of u v* · *u* $u^{-1>}w$] *lq-rq-reassoc-suf*[*of u w v* · *u*, *unfolded rq-triv*[*of v u*]]
by *auto*

lemmas *rq-prod* = *lq-prod*[*reversed*]

2.7 Equidivisibility

Equidivisibility is the following property: if

$$xy = uv,$$

then there exists a word t such that $xt = u$ and $ty = v$, or $ut = x$ and $y = tv$. For monoids over words, this property is equivalent to the freeness of the monoid. As the monoid of all words is free, we can prove that it is equidivisible. Related lemmas based on this property follow.

lemma *eqd*: $x \cdot y = u \cdot v \Longrightarrow |x| \leq |u| \Longrightarrow \exists\ t.\ x \cdot t = u \wedge t \cdot v = y$
by (*simp add*: *append-eq-conv-conj*)

lemma *eqdE*: **assumes** $x \cdot y = u \cdot v$ **and** $|x| \leq |u|$
obtains t **where** $x \cdot t = u$ **and** $t \cdot v = y$
using *eqd*[*OF assms*] **by** *blast*

lemma *eqdE'*: **assumes** $x \cdot y = u \cdot v$ **and** $|v| \leq |y|$
obtains t **where** $x \cdot t = u$ **and** $t \cdot v = y$
using *eqdE*[*OF assms(1)*] *lenarg*[*OF assms(1)*, *unfolded length-append*] *assms(2)*
by *auto*

lemma *eqd-pref*: $x \cdot y = u \cdot v \Longrightarrow |x| \leq |u| \Longrightarrow x \cdot (x^{-1>}u) = u \wedge (x^{-1>}u) \cdot v = y$
using *eqd lq-triv* **by** *blast*

lemma *eqd-prefE*: **assumes** $x \cdot y = u \cdot v$ **and** $|x| \leq |u|$
obtains t **where** $x \cdot t = u$ **and** $t \cdot v = y$
using *eqd-pref assms* **by** *blast*

lemma *eqd-pref1*: $x \cdot y = u \cdot v \Longrightarrow |x| \leq |u| \Longrightarrow x \cdot (x^{-1>}u) = u$
using *eqd-pref* **by** *blast*

lemma *eqd-pref2*: $x \cdot y = u \cdot v \Longrightarrow |x| \leq |u| \Longrightarrow (x^{-1>}u) \cdot v = y$
using *eqd-pref* **by** *blast*

lemma *eqd-equal*: $x \cdot y = u \cdot v \Longrightarrow |x| = |u| \Longrightarrow x = u \wedge y = v$
by *simp*

lemma *pref-equal*: $u \leq p\ v \cdot w \Longrightarrow |u| = |v| \Longrightarrow u = v$
by *simp*

lemma *eqd-equal-suf*: $x \cdot y = u \cdot v \Longrightarrow |y| = |v| \Longrightarrow x = u \wedge y = v$
by *simp*

lemma *eqd-comp*: **assumes** $x \cdot y = u \cdot v$ **shows** $x \bowtie u$
using *le-cases*[*of* $|x|$ $|u|$ $x \bowtie u$]
eqd-pref1[*of x y u v*, *THEN prefI*[*of x* $x^{-1>}u$ *u*], *OF assms*]
eqd-pref1[*of u v x y*, *THEN prefI*[*of u* $u^{-1>}x$ *x*], *OF assms*[*symmetric*]] **by** *auto*

— not equal to eqd_pref1[reversed]
lemma *eqd-suf1*: $x \cdot y = u \cdot v \Longrightarrow |x| \leq |u| \Longrightarrow (y^{<-1}v) \cdot v = y$
using *eqd-pref2 rq-triv* **by** *blast*

— not equal to eqd_pref2[reversed]
lemma *eqd-suf2*: **assumes** $x \cdot y = u \cdot v$ $|x| \leq |u|$ **shows** $x \cdot (y^{<-1}v) = u$
using *rq-reassoc*[*OF sufI*[*OF eqd-suf1*[*OF* ‹$x \cdot y = u \cdot v$› ‹$|x| \leq |u|$›]], *of x*, *unfolded* ‹$x \cdot y = u \cdot v$› *rq-triv*[*of u v*]]**.**

— not equal to eqd_pref[reversed]
lemma *eqd-suf*: **assumes** $x \cdot y = u \cdot v$ **and** $|x| \leq |u|$
shows $(y^{<-1}v) \cdot v = y \wedge x \cdot (y^{<-1}v) = u$
using *eqd-suf1*[*OF assms*] *eqd-suf2*[*OF assms*] **by** *blast*

2.8 Longest common prefix

notation *longest-common-prefix* (- $\wedge_p$ - [*61*,*62*] *64*) — provided by Sublist.thy

lemmas *lcp-simps* = *longest-common-prefix.simps* — provided by Sublist.thy

lemma *lcp-sym*: $u \wedge_p v = v \wedge_p u$
by (*induct u v rule*: *list-induct2′*) *auto*

— provided by Sublist.thy
lemmas *lcp-pref* = *longest-common-prefix-prefix1*
lemmas *lcp-pref′* = *longest-common-prefix-prefix2*
lemmas *pref-pref-lcp* = *longest-common-prefix-max-prefix*

lemma *lcp-take-eq*: *take* $(|u \wedge_p v|)$ *u* = *take* $(|u \wedge_p v|)$ *v*
using *pref-take*[*OF lcp-pref*[*of u v*]] *pref-take*[*OF lcp-pref′*[*of u v*]] **by** *simp*

lemma *lcp-pref-conv*: $u \wedge_p v = u \longleftrightarrow u \leq p\ v$
unfolding *prefix-order.eq-iff*[*of* $u \wedge_p v$ *u*]
using *lcp-pref′*[*of u v*]

lcp-pref[*of u v*] *longest-common-prefix-max-prefix*[*OF self-pref*[*of u*], *of v*]
by *auto*

lemma *pref-lcp-pref*: $w \leq_p u \wedge_p v \Longrightarrow w \leq_p u$
using *lcp-pref pref-trans* **by** *blast*

lemma *pref-lcp-pref'*: $w \leq_p u \wedge_p v \Longrightarrow w \leq_p v$
using *pref-lcp-pref*[*of w v u*, *unfolded lcp-sym*[*of v u*]]**.**

lemma *lcp-self*[*simp*]: $w \wedge_p w = w$
using *lcp-pref-conv* **by** *blast*

lemma *lcp-eq*: $|u| = |u \wedge_p v| \Longrightarrow u = u \wedge_p v$
using *long-pref*[*OF lcp-pref*, *of u v*] **by** *auto*

lemma *lcp-len*: $|u| \leq |u \wedge_p v| \Longrightarrow u \leq_p v$
using *long-pref*[*OF lcp-pref*, *of u v*] **unfolding** *lcp-pref-conv*[*symmetric*]**.**

lemma *lcp-len'*: $\neg\ u \leq_p v \Longrightarrow |u \wedge_p v| < |u|$
using *not-le-imp-less*[*OF contrapos-nn*[*OF - lcp-len*]]**.**

lemma *incomp-lcp-len*: **assumes** $\neg\ u \bowtie v$ **shows** $|u \wedge_p v| < min\ |u|\ |v|$
unfolding *min-less-iff-conj*[*of* $|u \wedge_p v|$ $|u|$ $|v|$]
using *assms lcp-len'*[*of u v*] *lcp-len'*[*of v u*, *folded lcp-sym*[*of u v*]]
min-less-iff-conj[*of* $|u \wedge_p v|$ $|u|$ $|v|$] **by** *blast*

lemma *lcp-ext-right* [*case-names comp non-comp*]: **obtains** $r \bowtie r' \mid (r \cdot u) \wedge_p (r' \cdot v) = r \wedge_p r'$
proof−
have $\neg\ r \bowtie r' \Longrightarrow r \cdot u \wedge_p r' \cdot v = r \wedge_p r'$
by (*induct r r' rule*: *list-induct2'*, *simp*+)
thus *?thesis*
using *that*(*1*) *that*(*2*) **by** *linarith*
qed

lemma *lcp-same-len*: $|u| = |v| \Longrightarrow u \neq v \Longrightarrow u \wedge_p v = u \cdot w \wedge_p v \cdot w'$
using *lcp-ext-right*[*of u v - w w'*] *pref-comp-eq*[*of u v*] **by** *argo*

lemma *lcp-mismatch*: $|u \wedge_p v| < |u| \Longrightarrow |u \wedge_p v| < |v| \Longrightarrow u!\ |u \wedge_p v| \neq v!\ |u \wedge_p v|$
by (*induct u v rule*: *list-induct2'*) *auto*

lemma *lcp-mismatch'*: **assumes** $\neg\ u \bowtie v$ **shows** $u!\ |u \wedge_p v| \neq v!\ |u \wedge_p v|$
using *incomp-lcp-len*[*OF assms*, *unfolded min-less-iff-conj*] *lcp-mismatch*
by *blast*

lemma *lcp-ext-left*: $(z \cdot u) \wedge_p (z \cdot v) = z \cdot (u \wedge_p v)$
by (*induct z*) *auto*

lemma *lcp-first-letters*: $u!0 \neq v!0 \Longrightarrow u \wedge_p v = \varepsilon$
by (*induct u v rule*: *list-induct2'*) *auto*

lemma *lcp-first-mismatch*: $a \neq b \Longrightarrow w \cdot [a] \cdot u \wedge_p w \cdot [b] \cdot v = w$
by (*simp add*: *lcp-ext-left*)

lemma *lcp-first-mismatch'*: $a \neq b \Longrightarrow u \cdot [a] \wedge_p u \cdot [b] = u$
using *lcp-first-mismatch*[*of a b u* ε ε] **by** *simp*

lemma *lcp-mismatch-shorter*: **assumes** $|u| = |v|$ $x \neq y$ **shows** $u \cdot [x] \wedge_p v \cdot [y] = u \wedge_p v$
by (*cases u* = *v*)
(*simp add*: *lcp-self*[*of v*] *lcp-first-mismatch'*[*OF* ‹$x \neq y$›, *of v*],
use lcp-same-len[*OF* ‹$|u| = |v|$›, *of* [*x*] [*y*]] **in** *auto*)

lemma *lcp-rulers*: $r \leq p\ s \Longrightarrow r' \leq p\ s' \Longrightarrow (r \bowtie r' \vee r \wedge_p r' = s \wedge_p s')$
using *lcp-ext-right prefD*[*of r s*] *prefD*[*of r' s'*] **by** *metis*

lemma *pref-pref-lcp'*: $w \leq p\ r \Longrightarrow w' \leq p\ s \Longrightarrow w \wedge_p w' \leq p\ (r \wedge_p s)$
using *pref-pref-lcp lcp-pref lcp-sym pref-trans* **by** *metis*

lemma *lcp-distinct-hd*: *hd u* $\neq$ *hd v* $\Longrightarrow u \wedge_p v = \varepsilon$
proof–
assume *hd u* $\neq$ *hd v*
hence $(u \neq \varepsilon \wedge v \neq \varepsilon) \Longrightarrow$ *hd u* $\neq$ *hd v* $\Longrightarrow u \wedge_p v = \varepsilon$
by (*simp add*: *lcp-first-letters hd-conv-nth*)
moreover have $u = \varepsilon \vee v = \varepsilon \Longrightarrow u \wedge_p v = \varepsilon$
using *lcp-pref'* **by** *auto*
ultimately show *?thesis* **using** ‹*hd u* $\neq$ *hd v*› **by** *blast*
qed

lemma *lcp-lenI*: **assumes** $i <$ *min* $|u|$ $|v|$ **and** *take i u* = *take i v* **and** $u!i \neq v!i$
shows $i = |u \wedge_p v|$
proof–
have *u*: *take i u* $\cdot$ [*u* ! *i*] $\cdot$ (*drop* (*Suc i*) *u*) = *u*
using ‹$i <$ *min* $|u|$ $|v|$› *id-take-nth-drop*[*of i u*] **by** *simp*
have *v*: *take i u* $\cdot$ [*v* ! *i*] $\cdot$ *drop* (*Suc i*) *v* = *v* **using** ‹$i <$ *min* $|u|$ $|v|$›
unfolding ‹*take i u* = *take i v*› **using** *id-take-nth-drop* **by** *fastforce*
from *lcp-first-mismatch*[*OF* ‹$u!i \neq v!i$›, *of take i u drop* (*Suc i*) *u drop* (*Suc i*) *v*, *unfolded u v*]
have $u \wedge_p v =$ *take i u*.
thus *?thesis*
using ‹$i <$ *min* $|u|$ $|v|$› **by** *auto*
qed

lemma *lcp-prefs*: $|u \cdot w \wedge_p v \cdot w'| < |u| \Longrightarrow |u \cdot w \wedge_p v \cdot w'| < |v| \Longrightarrow u \wedge_p v = u \cdot w \wedge_p v \cdot w'$
by (*induct u v rule*: *list-induct2'*) *auto*

2.8.1 Longest common prefix and prefix comparability

lemma *lexord-cancel-right*: $(u \cdot z,\ v \cdot w) \in \mathit{lexord}\ r \Longrightarrow \neg\ u \bowtie v \Longrightarrow (u,v) \in \mathit{lexord}\ r$
by (*induction rule*: *list-induct2'*, *simp+*, *auto*)

lemma *lcp-ruler*: $r \bowtie w1 \Longrightarrow r \bowtie w2 \Longrightarrow \neg\ w1 \bowtie w2 \Longrightarrow r \leq_p w1 \wedge_p w2$
unfolding *prefix-comparable-def*
by (*meson pref-pref-lcp pref-trans ruler*)

lemma *comp-monotone*: $w \bowtie r \Longrightarrow u \leq_p w \Longrightarrow u \bowtie r$
using *pref-trans*[*of u w r*] *ruler*[*of u w r*] **by** *blast*

lemma *comp-monotone'*: $w \bowtie r \Longrightarrow w \wedge_p w' \bowtie r$
using *comp-monotone*[*of w r w* $\wedge_p$ *w'*, *OF - longest-common-prefix-prefix1*].

lemma *double-ruler*: **assumes** $w \bowtie r$ **and** $w' \bowtie r'$ **and** $\neg\ r \bowtie r'$
shows $w \wedge_p w' \leq_p r \wedge_p r'$
using *comp-monotone'*[*OF* ‹$w' \bowtie r'$›, *of w*] **unfolding** *lcp-sym*[*of w' w*]
using *lcp-ruler*[*OF comp-monotone'*[*OF* ‹$w \bowtie r$›, *of w'*] - ‹$\neg\ r \bowtie r'$›] **by** *blast*

lemma *pref-comp-ruler*: **assumes** $w \bowtie u \cdot [x]$ **and** $w \bowtie v \cdot [y]$ **and** $x \neq y$ **and** $|u| = |v|$
shows $w \leq_p u \wedge w \leq_p v$
using *double-ruler*[*OF* ‹$w \bowtie u \cdot [x]$› ‹$w \bowtie v \cdot [y]$› *mismatch-incopm*[*OF* ‹$|u| = |v|$› ‹$x \neq y$›]] **unfolding** *lcp-self*[*of w*] *lcp-mismatch-shorter*[*OF* ‹$|u| = |v|$› ‹$x \neq y$›]
using *pref-lcp-pref pref-lcp-pref'* **by** *blast*

lemmas *suf-comp-ruler* = *pref-comp-ruler*[*reversed*]

2.9 Mismatch

The first pair of letters on which two words/lists disagree

fun *mismatch-pair* :: *'a list* ⇒ *'a list* ⇒ (*'a* × *'a*) **where**
mismatch-pair ε *v* = (ε!*0*, *v*!*0*) |
mismatch-pair v ε = (*v*!*0*, ε!*0*) |
mismatch-pair (*a*#*u*) (*b*#*v*) = (*if a*=*b then mismatch-pair u v else* (*a*,*b*))

Alternatively, mismatch pair may be defined using the longest common prefix as follows.

lemma *mismatch-pair-lcp*: *mismatch-pair u v* = (*u*!|*u*∧$_p$*v*|,*v*!|*u*∧$_p$*v*|)
proof(*induction u v rule*: *mismatch-pair.induct*, *simp+*)
qed

For incomparable words the pair is out of diagonal.

lemma *incomp-neq*: $\neg\ u \bowtie v \Longrightarrow (\mathit{mismatch\text{-}pair}\ u\ v) \notin \mathit{Id}$
unfolding *mismatch-pair-lcp* **by** (*simp add*: *lcp-mismatch'*)

lemma *mismatch-ext-left*: $\neg\ u \bowtie v \Longrightarrow$ *mismatch-pair u v* = *mismatch-pair* $(p{\cdot}u)$ $(p{\cdot}v)$
unfolding *mismatch-pair-lcp* **by** (*simp add*: *lcp-ext-left*)

lemma *mismatch-ext-right*: **assumes** $\neg\ u \bowtie v$
shows *mismatch-pair u v* = *mismatch-pair* $(u{\cdot}z)$ $(v{\cdot}w)$
proof–
have *less1*: $|u \wedge_p v| < |u|$ **and** *less2*: $|v \wedge_p u| < |v|$
using *lcp-len'*[*of u v*] *lcp-len'*[*of v u*] *assms* **by** *auto*
show *?thesis*
unfolding *mismatch-pair-lcp* **unfolding** *pref-index*[*OF triv-pref less1*, *of z*] *pref-index*[*OF triv-pref less2*, *of w*, *unfolded lcp-sym*[*of v*]]
using *assms lcp-ext-right*[*of u v - z w*] **by** *metis*
qed

lemma *mismatchI*: $\neg\ u \bowtie v \Longrightarrow i < min\ |u|\ |v| \Longrightarrow$ *take i u* = *take i v* $\Longrightarrow u!i \neq v!i$
$\Longrightarrow$ *mismatch-pair u v* = $(u!i,v!i)$
unfolding *mismatch-pair-lcp* **using** *lcp-lenI* **by** *blast*

For incomparable words, the mismatch letters work in a similar way as the lexicographic order

lemma *mismatch-lexord*: **assumes** $\neg\ u \bowtie v$ **and** *mismatch-pair u v* $\in r$
shows $(u,v) \in$ *lexord r*
unfolding *lexord-take-index-conv mismatch-pair-lcp*
using ‹*mismatch-pair u v* $\in r$›[*unfolded mismatch-pair-lcp*]
incomp-lcp-len[*OF assms*(*1*)] *lcp-take-eq* **by** *blast*

However, the equivalence requires r to be irreflexive. (Due to the definition of lexord which is designed for irreflexive relations.)

lemma *lexord-mismatch*: **assumes** $\neg\ u \bowtie v$ **and** *irrefl r*
shows *mismatch-pair u v* $\in r \longleftrightarrow (u,v) \in$ *lexord r*
proof
assume $(u,v) \in$ *lexord r*
obtain *i* **where** $i < min\ |u|\ |v|$ **and** *take i u* = *take i v* **and** $(u\ !\ i,\ v\ !\ i) \in r$
using ‹$(u,v) \in$ *lexord r*›[*unfolded lexord-take-index-conv*] ‹$\neg\ u \bowtie v$› *pref-take-conv* **by** *blast*
have $u!i \neq v!i$
using ‹*irrefl r*›[*unfolded irrefl-def*] ‹$(u\ !\ i,\ v\ !\ i) \in r$› **by** *fastforce*
from ‹$(u\ !\ i,\ v\ !\ i) \in r$›[*folded mismatchI*[*OF* ‹$\neg\ u \bowtie v$› ‹$i < min\ |u|\ |v|$› ‹*take i u* = *take i v*› ‹$u!i \neq v!i$›]]
show *mismatch-pair u v* $\in r$.
next
from *mismatch-lexord*[*OF* ‹$\neg\ u \bowtie v$›]
show *mismatch-pair u v* $\in r \Longrightarrow (u,\ v) \in$ *lexord r*.
qed

2.10 Factor properties

lemma *rev-fac*[*reversal-rule*]: $rev\ u \leq_f rev\ v \longleftrightarrow u \leq_f v$
using *Sublist.sublist-rev*.

lemma *fac-pref*: $u \leq_f v \equiv \exists\ p.\ p \cdot u \leq_p v$
by *simp*

lemma *fac-pref-suf*: $u \leq_f v \Longrightarrow \exists\ p.\ p \leq_p v \wedge u \leq_s p$
using *sublist-altdef* **by** *blast*

lemma *pref-suf-fac*: $r \leq_p v \Longrightarrow u \leq_s r \Longrightarrow u \leq_f v$
using *sublist-altdef* **by** *blast*

lemmas
$fac\text{-}suf = fac\text{-}pref[reversed]$ **and**
$fac\text{-}suf\text{-}pref = fac\text{-}pref\text{-}suf[reversed]$ **and**
$suf\text{-}pref\text{-}fac = pref\text{-}suf\text{-}fac[reversed]$

lemma *suf-pref-eq*: $s \leq_s p \Longrightarrow p \leq_p s \Longrightarrow p = s$
using *long-pref suffix-length-le* **by** *blast*

lemma *fac-triv'*: **assumes** $p{\cdot}x{\cdot}q = x$ **shows** $q = \varepsilon$
using *prefI*[*of* $p{\cdot}x\ q\ p{\cdot}x{\cdot}q$] *sufI*[*of* $\varepsilon\ p{\cdot}x{\cdot}q\ x$, *THEN suf-ext*[*of* $p{\cdot}x{\cdot}q\ x\ p$]]
suf-pref-eq[*of* $x\ p{\cdot}x$] *self-append-conv*[*of* $p{\cdot}x\ q$]
unfolding *assms append-Nil rassoc*
by *blast*

lemma *fac-triv*: $p{\cdot}x{\cdot}q = x \Longrightarrow p = \varepsilon$
using *fac-triv'* **by** *force*

lemmas
$suf\text{-}fac = suffix\text{-}imp\text{-}sublist$ **and**
$pref\text{-}fac = prefix\text{-}imp\text{-}sublist$

lemma *fac-Cons-E*: **assumes** $u \leq_f (a\#v)$
obtains $u \leq_p (a\#v) \mid u \leq_f v$
using *assms*[*unfolded sublist-Cons-right*]
by *fast*

lemmas
$fac\text{-}snoc\text{-}E = fac\text{-}Cons\text{-}E[reversed]$

2.11 Power and its properties

Word powers are often investigated in Combinatorics on Words. We thus interpret words as *monoid-mult* and adopt a notation for the word power.

declare *power.power.simps* [*code*]

interpretation *monoid-mult* ε *append*
by *standard simp+*

notation *power* (**infixr** $^{@}$ *80*)

— inherited power properties

lemma *pow-zero* [*simp*]: $u^{@}0 = \varepsilon$
using *power.power-0*.

lemma *emp-pow*: $\varepsilon^{@}n = \varepsilon$
using *power-one*.

lemma *pow-Suc-list*: $u^{@}(Suc\ n) = u \cdot u^{@}n$
using *power.power-Suc*.

lemma *pow-commutes-list*: $u^{@}n \cdot u = u \cdot u^{@}n$
using *power-commutes*.

lemma *pow-add-list*: $x^{@}(a+b) = x^{@}a \cdot x^{@}b$
using *power-add*.

lemma *pow-Suc2-list*: $u^{@}Suc\ n = u^{@}n \cdot u$
using *power-Suc2*.

lemma *pow-eq-if-list*: $p^{@}m = (if\ m = 0\ then\ \varepsilon\ else\ p \cdot p^{@}(m-1))$
using *power-eq-if*.

lemma *pow-one-id*: $u^{@}1 = u$
using *power-one-right*.

lemma *pow2-list*: $u^{@}2 = u \cdot u$
using *power2-eq-square*.

lemma *comm-add-exp*: $u \cdot v = v \cdot u \Longrightarrow u\ ^{@}\ n \cdot v = v \cdot u\ ^{@}\ n$
using *power-commuting-commutes*.

lemma *pow-mult-list*: $u^{@}(m*n) = (u^{@}m)^{@}n$
using *power-mult*.

lemma *pow-rev-emp-conv*[*reversal-rule*]: *power.power* $(rev\ \varepsilon)\ (\cdot) = (^{@})$
by *simp*

— more power properties

lemma *zero-exp*: $n = 0 \Longrightarrow r^{@}n = \varepsilon$
by *simp*

lemma *nemp-pow*[*elim*]: $t^{@}m \neq \varepsilon \Longrightarrow m \neq 0$
using *zero-exp* **by** *blast*

lemma *nemp-pow'*[*elim*]: $t^{@}m \neq \varepsilon \Longrightarrow t \neq \varepsilon$
using *emp-pow* **by** *auto*

lemma *sing-pow*:$i < m \Longrightarrow ([a]^{@}m)\ !\ i = a$
by (*induct i m rule*: *diff-induct*) *auto*

lemma *pow-is-concat-replicate*: $u^{@}n = concat\ (replicate\ n\ u)$
by (*induct n*) *auto*

lemma *pow-slide*: $u \cdot (v \cdot u)^{@}n\ \cdot v = (u \cdot v)^{@}(Suc\ n)$
by (*induct n*) *simp+*

lemma *pop-pow-one*: $m \neq 0 \Longrightarrow r^{@}m = r \cdot r^{@}(m-1)$
by (*simp add*: *pow-eq-if-list*)

lemma *hd-pow*: **assumes** $n \neq 0$ **shows** $hd(u^{@}n) = hd\ u$
unfolding *pop-pow-one*[*OF* ‹$n \neq 0$›] **using** *hd-append2* **by** (*cases* $u = \varepsilon$, *simp*)

lemma *pop-pow*: $m \leq k \Longrightarrow u^{@}m \cdot u^{@}(k-m) = u^{@}k$
using *le-add-diff-inverse pow-add-list* **by** *metis*

lemma *pop-pow-cancel*: $u^{@}k \cdot v = u^{@}m \cdot w \Longrightarrow m \leq k \Longrightarrow u^{@}(k-m) \cdot v = w$
using *lassoc pop-pow*[*of m k u*] *same-append-eq*[*of* $u^{@}m$ $u^{@}(k-m)\cdot v$ *w*, *unfolded lassoc*] **by** *argo*

lemma *pow-comm*: $t^{@}k \cdot t^{@}m = t^{@}m \cdot t^{@}k$
unfolding *pow-add-list*[*symmetric*] *add.commute*[*of k*]**..**

lemma *comm-add-exps*: **assumes** $r \cdot u = u \cdot r$ **shows** $r^{@}m \cdot u^{@}k = u^{@}k \cdot r^{@}m$
using *comm-add-exp*[*OF comm-add-exp*[*OF assms*, *symmetric*], *symmetric*]**.**

lemma *rev-pow*: $rev\ (x^{@}m) = (rev\ x)^{@}m$
by (*induct m*, *simp*, *simp add*: *pow-commutes-list*)

lemmas [*reversal-rule*] = *rev-pow*[*symmetric*]

lemmas *pow-eq-if-list'* = *pow-eq-if-list*[*reversed*] **and**
pop-pow-one' = *pop-pow-one*[*reversed*] **and**
pop-pow' = *pop-pow*[*reversed*] **and**
pop-pow-cancel' = *pop-pow-cancel*[*reversed*]

lemma *pow-len*: $|u^{@}k| = k * |u|$
by (*induct k*) *simp+*

lemma *eq-pow-exp*: **assumes** $u \neq \varepsilon$ **shows** $u^{@}k = u^{@}m \longleftrightarrow k = m$

proof
assume $k = m$ **thus** $u^{@}k = u^{@}m$ **by** *simp*
next
assume $u^{@}k = u^{@}m$
from *lenarg*[*OF this*, *unfolded pow-len mult-cancel2*]
show $k = m$
using ‹$u \neq \varepsilon$›[*folded length-0-conv*] **by** *blast*
qed

lemma *nemp-emp-power*: **assumes** $u \neq \varepsilon$ **shows** $u^{@}m = \varepsilon \longleftrightarrow m = 0$
using *eq-pow-exp*[*OF assms*] **by** *fastforce*

lemma *nonzero-pow-emp*: **assumes** $m \neq 0$ **shows** $u^{@}m = \varepsilon \longleftrightarrow u = \varepsilon$
by (*meson assms nemp-emp-power nemp-pow'*)

lemma *pow-eq-eq*:
assumes $u^{@}k = v^{@}k$ **and** $k \neq 0$
shows $u = v$
proof–
have $|u| = |v|$
using *lenarg*[*OF* ‹$u^{@}k = v^{@}k$›, *unfolded pow-len*] ‹$k \neq 0$› **by** *simp*
from *eqd-equal*[*of u* $u^{@}(k-1)$ *v* $v^{@}(k-1)$, *OF - this*]
show *?thesis*
using ‹$u^{@}k = v^{@}k$› **unfolding** *pop-pow-one*[*OF* ‹$k \neq 0$›] **by** *blast*
qed

lemma *sing-pow-empty*: $[a]^{@}n = \varepsilon \longleftrightarrow n = 0$
by (*simp add*: *nemp-emp-power*)

lemma *sing-pow-lists*: $a \in A \Longrightarrow [a]^{@}n \in lists\ A$
by (*induct n*, *auto*)

lemma *long-power*: $r \neq \varepsilon \Longrightarrow |x| \leq |r^{@}|x||$
unfolding *pow-len*[*of r* $|x|$] **using** *nemp-pos-len* **by** *auto*

lemma *long-power'*: $r \neq \varepsilon \Longrightarrow |x| < |r^{@}(Suc\ |x|)|$
unfolding *pow-Suc-list length-append*
by (*simp add*: *long-power add-strict-increasing*)

lemma *long-pow-exp*: $r \neq \varepsilon \Longrightarrow m \leq |r^{@}m|$
unfolding *pow-len*[*of r m*] **using** *nemp-pos-len*[*of r*] **by** *simp*

lemma *long-pow-ex*: **assumes** $r \neq \varepsilon$ **obtains** n **where** $m \leq |r^{@}n|$ **and** $n \neq 0$
proof–
obtain $x :: 'a\ list$ **where** $|x| = m$
using *Ex-list-of-length* **by** *auto*
show *thesis*
using *that*[*of m*, *OF long-power*[*OF* ‹$r \neq \varepsilon$›, *of x*, *unfolded* ‹$|x| = m$›]] *that*[*of*

Suc 1] **by** *auto*
qed

lemma *pref-pow-ext*: $x \leq_p r^{@}k \Longrightarrow x \leq_p r^{@}Suc\ k$
using *pref-trans*[*OF - prefI*[*OF pow-Suc2-list*[*symmetric*]]]**.**

lemma *pref-pow-ext'*: $u \leq_p r^{@}k \Longrightarrow u \leq_p r \cdot r^{@}k$
using *pref-pow-ext*[*unfolded pow-Suc-list*]**.**

lemma *pref-pow-root-ext*: $x \leq_p r^{@}k \Longrightarrow r \cdot x \leq_p r^{@}Suc\ k$
by *simp*

lemma *pref-prod-root*: $u \leq_p r^{@}k \Longrightarrow u \leq_p r \cdot u$
using *pref-pow-ext'*[*THEN pref-prod-pref*]**.**

lemma *pref-exps-pow*: $k \leq l \Longrightarrow r^{@}k \leq_p r^{@}l$
using *leI pop-pow*[*of k l r*] **by** *blast*

lemma *pref-exp-le*: **assumes** $u \neq \varepsilon$ $u^{@}m \leq_p u^{@}n$ **shows** $m \leq n$
using *mult-cancel-le*[*OF nemp-len*[*OF* ‹$u \neq \varepsilon$›], *of m n*]
prefix-length-le[*OF* ‹$u^{@}m \leq_p u^{@}n$›, *unfolded pow-len*[*of u m*] *pow-len*[*of u n*]]
by *blast*

lemmas
suf-pow-ext = *pref-pow-ext*[*reversed*] **and**
suf-pow-ext'= *pref-pow-ext'*[*reversed*] **and**
suf-pow-root-ext = *pref-pow-root-ext*[*reversed*] **and**
suf-prod-root = *pref-prod-root*[*reversed*] **and**
suf-exps-pow = *pref-exps-pow*[*reversed*] **and**
suf-exp-le = *pref-exp-le*[*reversed*]

lemma *comm-common-power*: **assumes** $r \cdot u = u \cdot r$ **shows** $r^{@}|u| = u^{@}|r|$
using *eqd-equal*[*OF comm-add-exps*[*OF* ‹$r \cdot u = u \cdot r$›], *of* $|u|$ $|r|$]
unfolding *pow-len* **by** *fastforce*

lemma *one-generated-list-power*: $u \in lists\ \{x\} \Longrightarrow \exists k.\ concat\ u = x^{@}k$
proof(*induction u*)
case *Nil*
then show *?case*
by (*simp add*: *pow-is-concat-replicate*)
next
case (*Cons a u*)
then show *?case*
unfolding *Cons-in-lists-iff concat.simps*(*2*)
using *singletonD*[*of a x*] *pow-Suc-list*[*of a*] **by** *metis*
qed

lemma *pow-lists*: $0 < k \Longrightarrow u^{@}k \in lists\ B \Longrightarrow u \in lists\ B$
by (*simp add*: *pow-eq-if-list*)

lemma *concat-morph-power*: $xs \in lists\ B \Longrightarrow xs = ts^{@}k \Longrightarrow concat\ ts^{@}k = concat\ xs$
by (*induct k arbitrary*: *xs ts*) *simp+*

lemma *pref-not-idem*: $z \neq \varepsilon \Longrightarrow z \neq x \Longrightarrow z \cdot x^{@}k \neq x$
using *fac-triv* **by** (*cases k*, *simp*, *auto*)

lemma *per-exp-pref*: $u \leq p\ r \cdot u \Longrightarrow u \leq p\ r^{@}k \cdot u$
proof(*induct k*, *simp*)
case (*Suc k*) **show** *?case*
unfolding *pow-Suc-list rassoc*
using *Suc.hyps Suc.prems pref-prolong* **by** *blast*
qed

lemmas
suf-not-idem = *pref-not-idem*[*reversed*] **and**
per-exp-suf = *per-exp-pref*[*reversed*]

lemma *hd-sing-power*: $k \neq 0 \Longrightarrow hd\ ([a]^{@}k) = a$
by (*induction k*) *simp+*

lemma *root-pref-cancel*: **assumes** $t^{@}m \cdot y = t^{@}k$ **shows** $y = t^{@}(k - m)$
using *lqI*[*of* $t^{@}m$ $t^{@}(k-m)$ $t^{@}k$] **unfolding** *lqI*[*OF* ‹$t^{@}m \cdot y = t^{@}k$›]
using *nat-le-linear*[*of m k*] *pop-pow*[*of m k t*] *diff-is-0-eq*[*of k m*] *append.right-neutral*[*of* $t^{@}k$] *pow-zero*[*of t*]
pref-antisym[*of* $t^{@}m$ $t^{@}k$, *OF prefI*[*OF* ‹$t^{@}m \cdot y = t^{@}k$›] *pref-exps-pow*[*of k m t*]]
by *presburger*

lemmas *root-suf-cancel* = *root-pref-cancel*[*reversed*]

lemma *index-pow-mod*: $i < |r^{@}k| \Longrightarrow (r^{@}k)!i = r!(i\ mod\ |r|)$
proof(*induction k*)
have *aux*: $|r^{@}(Suc\ l)| = |r^{@}l| + |r|$ **for** *l*
by *simp*
have *aux1*: $|(r^{@}l)| \leq i \Longrightarrow i < |r^{@}l| + |r| \Longrightarrow i\ mod\ |r| = i - |r^{@}l|$ **for** *l*
unfolding *pow-len*[*of r l*] **using** *less-diff-conv2*[*of* $l * |r|$ *i* $|r|$, *unfolded add.commute*[*of* $|r|$ $l * |r|$]]
get-mod[*of* $i - l * |r|$ $|r|$ *l*] *le-add-diff-inverse*[*of* $l*|r|$ *i*] **by** *argo*
case (*Suc k*)
show *?case*
unfolding *aux sym*[*OF pow-Suc2-list*[*symmetric*]] *nth-append le-mod-geq*
using *aux1*[*OF - Suc.prems*[*unfolded aux*]]
Suc.IH pow-Suc2-list[*symmetric*] *Suc.prems*[*unfolded aux*] *leI*[*of i* $|r\ ^{@}\ k|$] **by** *presburger*
qed *auto*

lemma *sing-pow-len*: $|[r]^{@}l| = l$

by (*induct l*) *auto*

lemma *concat-take-sing*: $k \leq l \Longrightarrow concat\ (take\ k\ ([r]^{@}l)) = r^{@}k$
proof(*induct k*, *simp*)
case (*Suc k*)
then show *?case*
using *concat-morph*[*of take k* $((r \# \varepsilon)\ ^{@}\ l)(r \# \varepsilon)$, *unfolded*
sym[*OF take-Suc-conv-app-nth*[*of k* $[r]^{@}l$, *unfolded sing-pow-len*[*of r l*]
less-eq-Suc-le
sing-pow[*OF iffD2*[*OF less-eq-Suc-le Suc.prems*], *of r*], *OF* ‹*Suc k* ≤ *l*›]]
concat-sing'[*of r*]
Suc.hyps[*OF Suc-leD*[*OF Suc.prems*]]
pow-Suc2-list[*symmetric*]]
by *argo*
qed

lemma *concat-sing-pow*: $concat\ ([a]^{@}k) = a^{@}k$
proof(*induct k*)
show $concat\ ((a \# \varepsilon)\ ^{@}\ 0) = a\ ^{@}\ 0$
by *simp*
next
fix *k* **assume** $concat\ ((a \# \varepsilon)\ ^{@}\ k) = a\ ^{@}\ k$
thus $concat\ ((a \# \varepsilon)\ ^{@}\ Suc\ k) = a\ ^{@}\ Suc\ k$
by *simp*
qed

lemma *unique-letter-word*: $(\forall\ c.\ c \in set\ w \longrightarrow c = a) \Longrightarrow \exists\ k.\ w = [a]^{@}k$
proof (*induct w*)
case *Nil*
then show *?case*
by (*metis pow-zero*)
next
case (*Cons b w*)
then show *?case*
proof−
obtain *k* **where** $w = [a]^{@}k$ **using** *Cons.hyps Cons.prems* **by** *auto*
hence $b\#w = [a]^{@}Suc\ k$
by (*simp add*: ‹$w = (a \# \varepsilon)^{@}k$› *Cons.prems*)
thus *?thesis* **by** *blast*
qed
qed

lemma *unique-letter-wordE*[*elim*]: **assumes** $(\forall\ c.\ c \in set\ w \longrightarrow c = a)$ **obtains** *k* **where** $w = [a]^{@}k$
using *unique-letter-word assms* **by** *metis*

lemma *unique-letter-wordE'*[*elim*]: **assumes** $set\ w \subseteq \{a\}$ **obtains** *k* **where** $w = [a]\ ^{@}\ k$
using *assms unique-letter-word*[*of w a*] **by** *blast*

lemma *conjug-pow*: $x \cdot z = z \cdot y \Longrightarrow x^{@}k \cdot z = z \cdot y^{@}k$
by (*induct k*) *fastforce+*

lemma *shift-pow*: $(u{\cdot}v)^{@}k{\cdot}u = u{\cdot}(v{\cdot}u)^{@}k$
by (*simp add*: *conjug-pow*)

lemma *lq-conjug-pow*: **assumes** $p \leq p\ x \cdot p$ **shows** $p^{-1>}(x^{@}k \cdot p) = (p^{-1>}(x \cdot p))^{@}k$
using *lqI*[*OF sym*[*OF conjug-pow*[*of x p* $p^{-1>}(x \cdot p)$, *OF sym*[*OF lq-pref*[*OF* ‹$p \leq p\ x \cdot p$›]], *of k*]]]**.**

lemmas *rq-conjug-pow* = *lq-conjug-pow*[*reversed*]

2.12 Total morphisms

locale *morphism* =
fixes *f* :: *'a list* ⇒ *'b list*
assumes *morph*: $f\ (u \cdot v) = f\ u \cdot f\ v$
begin

lemma *emp-to-emp*[*simp*]: $f\ \varepsilon = \varepsilon$
using *morph*[*of* ε ε] *self-append-conv2*[*of f* ε *f* ε] **by** *simp*

lemma *pow-morph*: $f\ (x^{@}k) = (f\ x)^{@}k$
by (*induction k*) (*simp add*: *morph*)+

lemma *pop-hd*: $f\ (a\#u) = f\ [a] \cdot f\ u$
unfolding *hd-word*[*of a u*] **using** *morph***.**

lemma *pop-hd-nemp*: $u \neq \varepsilon \Longrightarrow f\ (u) = f\ [hd\ u] \cdot f\ (tl\ u)$
using *list.exhaust-sel pop-hd*[*of hd u tl u*] **by** *force*

lemma *pop-last-nemp*: $u \neq \varepsilon \Longrightarrow f\ (u) = f\ (butlast\ u) \cdot f\ [last\ u]$
unfolding *morph*[*symmetric*] *append-butlast-last-id* **..**

lemma *pref-mono*: $u \leq p\ v \Longrightarrow f\ u \leq p\ f\ v$
using *morph* **by** *auto*

lemma *morph-concat-map*: $f\ x = concat\ (map\ (\lambda\ x.\ f\ [x])\ x)$
proof (*induction x*, *simp*)
case (*Cons a x*)
then show *?case*
unfolding *pop-hd*[*of a x*] **by** *auto*
qed

end

locale *two-morphisms* = *morphg*: *morphism g* + *morphh*: *morphism h* **for** *g h* ::

'a list ⇒ 'b list
begin
lemma *def-on-sings*:
assumes $\bigwedge a.\ g\ [a] = h\ [a]$
shows $g\ u = h\ u$
proof (*induct u*, *simp*)
next
case (*Cons a u*)
then show *?case*
unfolding *morphg.pop-hd*[*of a u*] *morphh.pop-hd*[*of a u*] **using** ‹$\bigwedge a.\ g\ [a] = h\ [a]$› **by** *presburger*
qed
end

2.13 Reversed morphism and composition

definition *rev-morph* :: (*'a list ⇒ 'b list*) ⇒ (*'a list ⇒ 'b list*) **where**
rev-morph f = *rev* ∘ *f* ∘ *rev*

lemma *rev-morph-idemp*[*simp*]: *rev-morph* (*rev-morph f*) = *f*
unfolding *rev-morph-def* **by** *auto*

lemma *morph-compose*: *morphism f* ⟹ *morphism g* ⟹ *morphism* (*f* ∘ *g*)
by (*simp add*: *morphism-def*)

lemma *rev-morph-arg*: *rev-morph f u* = *rev* (*f* (*rev u*))
by (*simp add*: *rev-morph-def*)

lemmas *rev-morph-arg-rev*[*reversal-rule*] = *rev-morph-arg*[*reversed add*: *rev-rev-ident*]

lemma *rev-morph-sing*: *rev-morph f* [*a*] = *rev* (*f* [*a*])
unfolding *rev-morph-def* **by** *simp*

context *morphism*
begin

lemma *rev-morph-morph*: *morphism* (*rev-morph f*)
by (*standard*, *auto simp add*: *rev-morph-def morph*)

lemma *morph-rev-len*: |*f* (*rev u*)| = |*f u*|
proof (*induction u*, *simp*)
case (*Cons a u*)
then show *?case*
unfolding *rev.simps*(*2*) *pop-hd*[*of a u*] *morph length-append* **by** *force*
qed

lemma *rev-morph-len*: |*rev-morph f u*| = |*f u*|
unfolding *rev-morph-def*
by (*simp add*: *morph-rev-len*)

end

2.14 Rotation

lemma *rotate-comp-eq*:$w \bowtie$ *rotate n w* $\Longrightarrow$ *rotate n w* = *w*
using *pref-same-len*[*OF - length-rotate*[*of n w*]] *pref-same-len*[*OF - length-rotate*[*of n w*, *symmetric*], *symmetric*]
by *blast*

corollary *mismatch-iff-lexord*: **assumes** *rotate n w* $\neq$ *w* **and** *irrefl r*
shows *mismatch-pair w* (*rotate n w*) $\in$ *r* $\longleftrightarrow$ (*w*,*rotate n w*) $\in$ *lexord r*
proof−
have ¬ *w* $\bowtie$ *rotate n w*
using *rotate-comp-eq* ‹*rotate n w* $\neq$ *w*›
unfolding *prefix-comparable-def* **by** *blast*
from *lexord-mismatch*[*OF this* ‹*irrefl r*›]
show *?thesis*.
qed

lemma *rotate-back*: **obtains** *m* **where** *rotate m* (*rotate n u*) = *u*
proof(*cases u* = ε, *simp*)
assume *u* $\neq$ ε
show *?thesis*
using *that*[*of* |*u*| − *n mod* |*u*|]
unfolding *rotate-rotate*[*of* |*u*| − *n mod* |*u*| *n mod* |*u*| *u*]
le-add-diff-inverse2[*OF*
less-imp-le-nat[*OF mod-less-divisor*[*OF nemp-len*[*OF* ‹*u* $\neq$ ε›, *unfolded neq0-conv*], *of n*]]]
arg-cong[*OF rotate-conv-mod*[*of n u*], *of rotate* (|*u*| − *n mod* |*u*|)]
by *simp*
qed

lemma *rotate-class-rotate*′: ($\exists$ *n*. *rotate n w* = *u*) $\longleftrightarrow$ ($\exists$ *n*. *rotate n* (*rotate l w*) = *u*)
proof
obtain *m* **where** *rot-m*: *rotate m* (*rotate l w*) = *w* **using** *rotate-back*.
assume $\exists$ *n*. *rotate n w* = *u*
then obtain *n* **where** *rot-n*: *rotate n w* = *u* **by** *blast*
show $\exists$ *n*. *rotate n* (*rotate l w*) = *u*
using *exI*[*of* λ *x*. *rotate x* (*rotate l w*) = *u n*+*m*, *OF*
rotate-rotate[*symmetric*, *of n m rotate l w*, *unfolded rot-m rot-n*]].
next
show $\exists$ *n*. *rotate n* (*rotate l w*) = *u* $\Longrightarrow$ $\exists$ *n*. *rotate n w* = *u*
using *rotate-rotate*[*symmetric*] **by** *blast*
qed

lemma *rotate-class-rotate*: {*u* . $\exists$ *n*. *rotate n w* = *u*} = {*u* . $\exists$ *n*. *rotate n* (*rotate l w*) = *u*}

using *rotate-class-rotate′* **by** *blast*

lemma *rotate-pow-self*: *rotate* $(l*|u|)$ $(u^{@}k) = u^{@}k$
proof(*induct l*, *simp*)
case (*Suc l*)
then show *?case*
proof(*cases k = 0*, *simp*)
assume $k \neq 0$
show *?thesis*
unfolding *rotate-rotate*[*of* $|u|$ $l * |u|$ $u^{@}k$, *unfolded mult-Suc*[*symmetric*] *Suc.hyps*, *symmetric*]
using *rotate-append*[*of u* $u^{@}(k-1)$, *folded pop-pow-one*[*OF* ‹$k \neq 0$›, *of u*] *pop-pow-one′*[*OF* ‹$k \neq 0$›, *of u*]].
qed
qed

lemma *rotate-root-self*: *rotate* $|r|$ $(r^{@}k) = r^{@}k$
using *rotate-pow-self*[*of 1 r k*] **by** *auto*

lemma *rotate-pow-mod*: *rotate n* $(u^{@}k)$ = *rotate* $(n \bmod |u|)$ $(u^{@}k)$
using *rotate-rotate*[*of n mod* $|u|$ *n div* $|u| * |u|$ $u^{@}k$, *symmetric*]
unfolding *rotate-pow-self*[*of n div* $|u|$ *u k*] *div-mult-mod-eq*[*of n* $|u|$, *unfolded add.commute*[*of n div* $|u| * |u|$ *n mod* $|u|$]].

lemma *rotate-conj-pow*: *rotate* $|u|$ $((u{\cdot}v)^{@}k) = (v{\cdot}u)^{@}k$
by (*induct k*, *simp*, *simp add*: *rotate-append shift-pow*)

lemma *rotate-pow-comm*: *rotate n* $(u^{@}k)$ = $(rotate\ n\ u)^{@}k$
proof (*cases* $u = \varepsilon$, *simp*)
assume $u \neq \varepsilon$
show *?thesis*
unfolding *rotate-drop-take*[*of n u*] *rotate-pow-mod*[*of n u k*]
using *rotate-conj-pow*[*of take* $(n \bmod |u|)$ *u drop* $(n \bmod |u|)$ *u k*, *unfolded append-take-drop-id*[*of n mod* $|u|$ *u*]]
unfolding *mod-le-divisor*[*of* $|u|$ *n*, *THEN take-len*, *OF* ‹$u \neq \varepsilon$›[*unfolded length-greater-0-conv*[*symmetric*]]].
qed

2.14.1 Lists of words and their concatenation

The helpful lemmas of this section deal with concatenation of a list of words *concat*. The main objective is to cover elementary facts needed to study factorizations of words.

lemma *append-in-lists*: $u \in lists\ A \Longrightarrow v \in lists\ A \Longrightarrow u \cdot v \in lists\ A$
by *simp*

lemma *concat-take-is-prefix*: *concat*(*take n ws*) $\leq_p$ *concat ws*
using *concat-morph*[*of take n ws drop n ws*, *unfolded append-take-drop-id*[*of n ws*], *THEN prefI*].

lemma *concat-take-suc*: **assumes** $j < |ws|$ **shows** $concat(take\ j\ ws) \cdot ws!j = concat(take\ (Suc\ j)\ ws)$
unfolding *take-Suc-conv-app-nth*[*OF* ‹$j < |ws|$›]
using *sym*[*OF concat-append*[*of* (*take j ws*) [*ws* ! *j*],
unfolded concat.simps(*2*)[*of ws*!*j* ε, *unfolded concat.simps*(*1*) *append-Nil2*]]].

lemma *pref-mod-list'*: **assumes** $u \leq np\ concat\ ws$
obtains *j r* **where** $j < |ws|$ **and** $r \leq np\ ws!j$ **and** $concat\ (take\ j\ ws) \cdot r = u$
proof−
have $|ws| \neq 0$
using *assms* **by** *fastforce*
then obtain *l* **where** $Suc\ l = |ws|$
using *Suc-pred* **by** *blast*
let $?P = \lambda\ j.\ u \leq p\ concat(take\ (Suc\ j)\ ws)$
have *?P l*
using *assms* ‹$Suc\ l = |ws|$› **by** *auto*

define *j* **where** $j = (LEAST\ j.\ ?P\ j)$ — largest j such that concat (take j ws) <p u
have $u \leq p\ concat(take\ (Suc\ j)\ ws)$ **and** $j < |ws|$
using *Least-le*[*of ?P*, *OF* ‹*?P l*›]
LeastI[*of ?P*, *OF* ‹*?P l*›] ‹$Suc\ l = |ws|$›
unfolding *sym*[*OF j-def*] **by** *auto*

have $concat(take\ j\ ws) <p\ u$
proof (*cases* $j = 0$)
assume $j = 0$ **thus** $concat(take\ j\ ws) <p\ u$
using *assms* **by** *fastforce*
next
assume $j \neq 0$ **hence** $j - 1 < j$ **and** $Suc\ (j-1) = j$ **by** *auto*
hence ¬ *?P* (*j*−*1*)
using *j-def not-less-Least* **by** *blast*
from *this*[*unfolded* ‹$Suc\ (j-1) = j$›]
show $concat(take\ j\ ws) <p\ u$
using *pref-comp-not-pref ruler*[*OF concat-take-is-prefix npD*[*OF assms*], *of j*]
unfolding *prefix-comparable-def* **by** *blast*
qed

let $?r = concat(take\ j\ ws)^{-1>}u$
have $concat(take\ j\ ws) \cdot ?r = u$ **and** $?r \neq \varepsilon$
using ‹$concat\ (take\ j\ ws) <p\ u$› **by** *auto*
have $?r \leq p\ ws!j$
using *pref-cancel*[*of concat* (*take j ws*) *concat* (*take j ws*)$^{-1>}$*u ws* ! *j*]
‹$u \leq p\ concat\ (take\ (Suc\ j)\ ws)$›[*unfolded sym*[*OF concat-take-suc*[*OF* ‹$j < |ws|$›]]]
‹$concat\ (take\ j\ ws) \cdot concat\ (take\ j\ ws)^{-1>}u = u$› **by** *argo*
show *thesis*
using *that*[*OF* ‹$j < |ws|$› *npI*[*OF* ‹*?r* ≠ ε› ‹*?r* ≤*p ws*!*j*›] ‹$concat(take\ j\ ws) \cdot ?r = u$›].

qed

lemma *pref-mod-list-suf*: **assumes** $u \leq_{np} concat\ ws$
obtains $j\ s$ **where** $j < |ws|$ **and** $s <_s ws!j$ **and** $concat\ (take\ (Suc\ j)\ ws) = u \cdot s$
proof–
obtain $j\ r$ **where** $j < |ws|$ **and** $r \leq_{np} ws!j$ **and** $concat\ (take\ j\ ws) \cdot r = u$
using *pref-mod-list'*[*OF assms*] **by** *blast*
have $r^{-1>}(ws!j) <_s (ws!j)$
using *ssufI1*[*OF - npD'*[*OF* ‹$r \leq_{np} ws\ !\ j$›]] *lq-pref*[*OF npD*[*OF* ‹$r \leq_{np} ws\ !\ j$›]].
have $concat\ (take\ (Suc\ j)\ ws) = u \cdot r^{-1>}(ws!j)$
using *lq-pref*[*OF npD*[*OF* ‹$r \leq_{np} ws\ !\ j$›], *symmetric*, *unfolded*
same-append-eq[*of concat* (*take j ws*) *ws* ! *j r* · $r^{-1>}$*ws* ! *j*, *symmetric*] *lassoc*
‹*concat* (*take j ws*) · *r* = *u*› *concat-take-suc*[*OF* ‹*j* < |*ws*|›] ‹$r \leq_{np} ws!j$›].
from *that*[*OF* ‹$j < |ws|$› ‹$r^{-1>}(ws!j) <_s (ws!j)$› *this*]
show *thesis*.
qed

lemma *suf-mod-list'*: $s \leq_{ns} concat\ ws \Longrightarrow ws \neq \varepsilon \Longrightarrow \exists j\ r.\ j < |ws| \wedge r \cdot (concat\ (drop\ (Suc\ j)\ ws)) = s \wedge r \leq_{ns} ws!j$
proof–
assume $s \leq_{ns} concat\ ws\ ws \neq \varepsilon$

have $rev\ s \leq_{np} concat\ (rev\ (map\ (\lambda u.\ rev\ u)\ ws))$
using ‹$s \leq_{ns} concat\ ws$›[*unfolded nsuf-rev-pref-iff*]
by (*simp add*: *rev-concat rev-map*)
have $rev\ ws \neq \varepsilon$
by (*simp add*: ‹$ws \neq \varepsilon$›)
then obtain $j\ r$ **where** $j < |rev\ (map\ rev\ ws)|$ $concat\ (take\ j\ (rev\ (map\ rev\ ws)))$ · $r = rev\ s$ $r \leq_{np} rev\ (map\ rev\ ws)\ !\ j$
using *pref-mod-list'*[*of rev s rev* (*map* (λ*u*. *rev u*) *ws*) *thesis*]
‹*rev s* ≤*np concat* (*rev* (*map rev ws*))› **by** *blast*

have $rev\ ws\ !\ j = rev\ (rev\ (map\ rev\ ws)\ !\ j)$
using ‹$j < |rev\ (map\ rev\ ws)|$› *length-map nth-map*[*of j rev ws rev*, *unfolded rev-map*[*of rev ws*, *symmetric*]] *rev-rev-ident*
by *simp*
from ‹$j < |rev\ (map\ rev\ ws)|$› *rev-nth*[*of j ws*, *unfolded this*]
have $rev\ (rev\ (map\ rev\ ws)\ !\ j) = ws!(|ws| - Suc\ j)$
by *fastforce*
from ‹$r \leq_{np} rev\ (map\ rev\ ws)\ !\ j$›[*unfolded npref-rev-suf-iff*, *unfolded this*]
have *p2*: $rev\ r \leq_{ns} ws!(|ws| - Suc\ j)$.
have $concat\ (take\ j\ (rev\ (map\ rev\ ws))) = rev\ (concat\ (drop\ (|ws| - j)\ ws))$
by (*simp add*: *rev-concat rev-map take-map take-rev*)
from *arg-cong*[*OF this*, *of* λ*x*. *x*·*r*, *unfolded* ‹*concat* (*take j* (*rev* (*map rev ws*))) · *r* = *rev s*›]
have *p1*: $s = rev\ r \cdot (concat\ (drop\ (|ws| - j)\ ws))$

using *rev-append*[*of rev* (*concat* (*drop* (|*ws*| − *j*) *ws*)) *r*] *rev-rev-ident*[*of s*] *rev-rev-ident*[*of* (*concat* (*drop* (|*ws*| − *j*) *ws*))]
by *argo*

have *p0*: |*ws*| − *Suc j* < |*ws*|
by (*simp add*: ‹*ws* ≠ *ε*›)

have |*ws*| − *j* = *Suc* (|*ws*| − *Suc j*)
using *Suc-diff-Suc*[*OF* ‹*j* < |*rev* (*map rev ws*)|›] **by** *auto*
from *p0 p1*[*unfolded this*] *p2* **show** *?thesis*
by *blast*
qed

lemma *pref-mod-list*: **assumes** *u* <*p concat ws*
obtains *j r* **where** *j* < |*ws*| **and** *r* <*p ws*!*j* **and** *concat* (*take j ws*) · *r* = *u*
proof−
have |*ws*| ≠ *0*
using *assms* **by** *auto*
then obtain *l* **where** *Suc l* = |*ws*|
using *Suc-pred* **by** *blast*
let *?P* = *λ j. u* <*p concat*(*take* (*Suc j*) *ws*)
have *?P l*
using *assms* ‹*Suc l* = |*ws*|› **by** *auto*
define *j* **where** *j* = (*LEAST j. ?P j*) — smallest j such that concat (take (Suc j) ws) <p u
have *u* <*p concat*(*take* (*Suc j*) *ws*)
using *LeastI*[*of ?P*, *OF* ‹*?P l*›] **unfolding** *sym*[*OF j-def*].
have *j* < |*ws*|
using *Least-le*[*of ?P*, *OF* ‹*?P l*›] ‹*Suc l* = |*ws*|› **unfolding** *sym*[*OF j-def*]
by *auto*
have *0* < *j* ⟹ *concat*(*take j ws*) ≤*p u*
using *Least-le*[*of ?P* (*j*− *Suc 0*), *unfolded sym*[*OF j-def*]]
ruler[*OF concat-take-is-prefix sprefD1*[*OF assms*], *of j*]
by *force*
hence *concat*(*take j ws*) ≤*p u*
by *fastforce*
let *?r* = *concat*(*take j ws*)$^{-1>}$*u*
have *concat*(*take j ws*) · *?r* = *u*
using ‹*concat* (*take j ws*) ≤*p u*› **by** *auto*
have *?r* <*p ws*!*j*
using *spref-lq*[*OF* ‹*concat* (*take j ws*) ≤*p u*› ‹*u* <*p concat* (*take* (*Suc j*) *ws*)›, *unfolded*
lq-triv[*of concat* (*take j ws*) *ws*!*j*, *unfolded concat-take-suc*[*OF* ‹*j* < |*ws*|›]]].
show *thesis*
using *that*[*OF* ‹*j* < |*ws*|› ‹*?r* <*p ws*!*j*› ‹*concat*(*take j ws*) · *?r* = *u*›].
qed

lemma *pref-mod-power*: **assumes** *u* <*p* $w^{@}l$
obtains *k z* **where** *k* < *l* **and** *z* <*p w* **and** $w^{@}k{\cdot}z$ = *u*

using *pref-mod-list*[*of u* $[w]^{@}l$, *unfolded sing-pow-len concat-sing-pow*, *OF* ‹$u <p w^{@}l$›, *of thesis*]
sing-pow[*of - l w*] *concat-take-sing*
less-imp-le-nat **by** *metis*

lemma *get-pow-exp*: **assumes** $z <p\ t$ **shows** $m = |t^{@}m{\cdot}z|\ div\ |t|$
unfolding *length-append*[*of* $t^{@}m\ z$, *unfolded pow-len*] **using** *get-div*[*OF prefix-length-less*[*OF assms*]].

lemma *get-pow-remainder*: **assumes** $z <p\ t$ **shows** $z = drop\ ((|t^{@}m{\cdot}z|\ div\ |t|){*}|t|)\ (t^{@}m{\cdot}z)$
using *drop-pref*[*of* $t^{@}m\ z$] *pow-len*[*of t m*] *get-pow-exp*[*OF assms*, *of m*] **by** *simp*

lemma *pref-mod-power′*: **assumes** $u \leq np\ w^{@}l$
obtains $k\ z$ **where** $k < l$ **and** $z \leq np\ w$ **and** $w^{@}k{\cdot}z = u$
using *pref-mod-list′*[*of u* $[w]^{@}l$, *unfolded sing-pow-len concat-sing-pow*, *OF* ‹$u \leq np\ w^{@}l$›]
sing-pow[*of - l w*] *concat-take-sing*[*of - l w*]
less-imp-le-nat[*of - l*] **by** *metis*

lemma *pref-power*: **assumes** $t \neq \varepsilon$ **and** $u \leq p\ t^{@}k$
shows $\exists\ m.\ t^{@}m \leq p\ u \wedge u <p\ t^{@}m \cdot t$
proof (*cases* $u = t^{@}k$)
show $u = t\ ^{@}\ k \Longrightarrow \exists m.\ t\ ^{@}\ m \leq p\ u \wedge u <p\ t\ ^{@}\ m \cdot t$
using ‹$t \neq \varepsilon$› **by** *blast*
next
assume $u \neq t\ ^{@}\ k$
obtain $m\ z$ **where** $m < k\ z <p\ t\ t\ ^{@}\ m \cdot z = u$
using *pref-mod-power*[*of u t k*] ‹$u \leq p\ t^{@}k$›[*unfolded prefix-order.dual-order.order-iff-strict*] ‹$u \neq t^{@}k$›
by *blast*
hence $t\ ^{@}\ m \leq p\ u$ **and** $u <p\ t\ ^{@}\ m \cdot t$ **by** *auto*
thus *?thesis* **by** *blast*
qed

lemma *pref-powerE*: **assumes** $t \neq \varepsilon$ **and** $u \leq p\ t^{@}k$
obtains m **where** $t^{@}m \leq p\ u\ u <p\ t^{@}m \cdot t$
using *assms pref-power* **by** *blast*

lemma *pref-power′*: **assumes** $u \neq \varepsilon$ **and** $u \leq p\ t^{@}k$
shows $\exists\ m.\ t^{@}m <p\ u \wedge u \leq p\ t^{@}m \cdot t$
proof−
obtain $m\ z$ **where** $m < k\ z \leq np\ t\ t\ ^{@}\ m \cdot z = u$
using *pref-mod-power′*[*OF npI*[*OF* ‹$u \neq \varepsilon$› ‹$u \leq p\ t^{@}k$›]].
thus *?thesis*
by *auto*
qed

lemma *suf-power*: **assumes** $t \neq \varepsilon$ **and** $u \leq s\ t^{@}k$ **shows** $\exists m.\ t^{@}m \leq s\ u \wedge u <s\ t$

$\cdot\ t^{@}m$
proof−
have *rev u* $\leq p$ *(rev t)*$^{@}k$
using ‹*u* $\leq s$ *t*$^{@}k$›[*unfolded suffix-to-prefix rev-pow*].
from *pref-power*[*OF - this*]
obtain *m* **where** *(rev t)*$^{@}m$ $\leq p$ *rev u* **and** *rev u* $<p$ *(rev t)*$^{@}m \cdot$ *rev t*
using ‹$t \neq \varepsilon$› **by** *blast*
have $t^{@}m \leq s$ *u*
using ‹*(rev t)*$^{@}m$ $\leq p$ *rev u*›[*folded suffix-to-prefix rev-pow*].
moreover have *u* $<s$ *t* $\cdot$ $t^{@}m$
using *sprefD1*[*OF* ‹*rev u* $<p$ *(rev t)*$^{@}m \cdot$ *rev t*›, *folded rev-pow rev-append suffix-to-prefix*]
sprefD2[*OF* ‹*rev u* $<p$ *(rev t)*$^{@}m \cdot$ *rev t*›, *folded rev-pow rev-append*]
by *blast*
ultimately show *?thesis* **by** *blast*
qed

lemma *suf-powerE*: **assumes** $t \neq \varepsilon$ **and** $u \leq s$ $t^{@}k$
obtains *m* **where** $t^{@}m \leq s$ *u* *u* $<s$ *t* $\cdot$ $t^{@}m$
using *assms suf-power* **by** *blast*

lemma *del-emp-concat*: *concat us* = *concat* (*filter* ($\lambda x.\ x \neq \varepsilon$) *us*)
by (*induct us*) *simp+*

lemma *lists-drop-emp*: *us* $\in$ *lists* $C_+ \Longrightarrow$ *us* $\in$ *lists C*
by *blast*

lemma *lists-drop-emp′*: *us* $\in$ *lists C* $\Longrightarrow$ (*filter* ($\lambda x.\ x \neq \varepsilon$) *us*) $\in$ *lists* C_+
by (*simp add*: *in-lists-conv-set*)

lemma *pref-concat-pref*: *us* $\leq p$ *ws* $\Longrightarrow$ *concat us* $\leq p$ *concat ws*
by *auto*

lemma *le-take-is-prefix*: **assumes** $k \leq n$ **shows** *take k ws* $\leq p$ *take n ws*
using *take-add*[*of k* (*n*−*k*) *ws*, *unfolded le-add-diff-inverse*[*OF* ‹$k \leq n$›]]
by *force*

lemma *take-pp-less*: **assumes** *take k ws* $<p$ *take n ws* **shows** $k < n$
using *conjunct2*[*OF sprefD*[*OF assms*]]
leI[*of k n*, *THEN*[*2*] *le-take-is-prefix*[*of n k ws*, *THEN*[*2*] *pref-antisym*[*of take k ws take n ws*]], *OF conjunct1*[*OF sprefD*[*OF assms*]]]
by *blast*

lemma *concat-pp-less*: **assumes** *concat* (*take k ws*) $<p$ *concat* (*take n ws*) **shows** $k < n$
using *le-take-is-prefix*[*of n k ws*, *THEN pref-concat-pref*] *conjunct1*[*OF sprefD*[*OF assms*]]
conjunct2[*OF sprefD*[*OF assms*]] *pref-antisym*[*of concat*(*take k ws*) *concat*(*take n ws*)]

by *fastforce*

2.15 Root

definition *root* :: *'a list* ⇒ *'a list* ⇒ *bool* (- ∈ -∗ [*51*,*51*] *60*)
where $u \in r* = (\exists\ k.\ r^{@}k = u)$
notation (*latex* **output**) *root* (- ∈ -*)

Empty word has all roots, including the empty root.

lemma $\varepsilon \in r*$
unfolding *root-def* **using** *power-0* **by** *blast*

lemma *rootI*: $r^{@}k \in r*$
using *root-def* **by** *auto*

lemma *self-root*: $u \in u*$
using *rootI*[*of u Suc 0*] **by** *simp*

lemma *rootE*: **assumes** $u \in r*$ **obtains** k **where** $r^{@}k = u$
using *assms root-def* **by** *auto*

lemma *empty-all-roots*[*simp*]: $\varepsilon \in r*$
using *rootI*[*of r 0*, *unfolded pow-zero*]**.**

lemma *take-root*: $k \neq 0 \Longrightarrow r = take\ (length\ r)\ (r^{@}k)$
by (*simp add*: *pow-eq-if-list*)

lemma *root-nemp*: $u \neq \varepsilon \Longrightarrow u \in r* \Longrightarrow r \neq \varepsilon$
unfolding *root-def* **using** *emp-pow* **by** *auto*

lemma *root-shorter*: $u \neq \varepsilon \Longrightarrow u \in r* \Longrightarrow u \neq r \Longrightarrow |r| < |u|$
by (*metis root-def leI take-all take-root pow-zero*)

lemma *root-shorter-eq*: $u \neq \varepsilon \Longrightarrow u \in r* \Longrightarrow |r| \leq |u|$
using *root-shorter* **by** *fastforce*

lemma *root-trans*[*trans*]: $[\![v \in u*;\ u \in t*]\!] \Longrightarrow v \in t*$
by (*metis root-def pow-mult-list*)

lemma *root-pow-root*[*trans*]: $v \in u* \Longrightarrow v^{@}n \in u*$
using *rootI root-trans* **by** *blast*

lemma *root-len*: $u \in q* \Longrightarrow \exists\, k.\ |u| = k*|q|$
unfolding *root-def* **using** *pow-len* **by** *auto*

lemma *root-len-dvd*: $u \in t* \Longrightarrow |t|\ dvd\ |u|$
using *root-len root-def* **by** *fastforce*

lemma *common-root-len-gcd*: $u \in t* \Longrightarrow v \in t* \Longrightarrow |t|\ dvd\ (\ gcd\ |u|\ |v|\ )$

by (*simp add*: *root-len-dvd*)

lemma *add-root*[*simp*]: $z \cdot w \in z* \longleftrightarrow w \in z*$
proof
assume $w \in z*$ **thus** $z \cdot w \in z*$
unfolding *root-def* **using** *pow-Suc-list* **by** *blast*
next
assume $z \cdot w \in z*$ **thus** $w \in z*$
unfolding *root-def*
using *root-pref-cancel*[*of z 1 w*, *unfolded power-one-right*] **by** *metis*
qed

lemma *add-roots*: $w \in z* \Longrightarrow w' \in z* \Longrightarrow w \cdot w' \in z*$
unfolding *root-def* **using** *pow-add-list* **by** *blast*

lemma *concat-sing-list-pow*: $ws \in lists\ \{u\} \Longrightarrow |ws| = k \Longrightarrow concat\ ws = u^{@}k$
proof(*induct k arbitrary*: *ws*)
case (*Suc k*)
have $ws \neq \varepsilon$
using *list.size*(*3*) *nat.distinct*(*2*)[*of k*, *folded* ‹$|ws| = Suc\ k$›] **by** *blast*
from *hd-Cons-tl*[*OF this*]
have $ws = hd\ ws \#\ tl\ ws$ **and** $|tl\ ws| = k$
using ‹ $|ws| = Suc\ k$› **by** *simp*+
then show *?case*
unfolding *pow-Suc-list hd-concat-tl*[*OF* ‹$ws \neq \varepsilon$›, *symmetric*]
using *Suc.hyps*[*OF tl-lists*[*OF* ‹ $ws \in lists\ \{u\}$›] ‹$|tl\ ws| = k$›]
Nitpick.size-list-simp(*2*) *lists-hd*[*of ws* $\{u\}$] ‹$ws \in lists\{u\}$› **by** *blast*
qed *simp*

lemma *concat-sing-list-pow'*: $ws \in lists\{u\} \Longrightarrow concat\ ws = u^{@}|ws|$
by (*simp add*: *concat-sing-list-pow*)

lemma *root-pref-cancel'*: **assumes** $x{\cdot}y \in t*$ **and** $x \in t*$ **shows** $y \in t*$
proof−
obtain *n m* **where** $t^{@}m = x \cdot y$ **and** $t^{@}n = x$
using ‹$x{\cdot}y \in t*$›[*unfolded root-def*] ‹$x \in t*$›[*unfolded root-def*] **by** *blast*
from *root-pref-cancel*[*of t n y m*, *unfolded this*]
show $y \in t*$
using *rootI* **by** *auto*
qed

lemma *root-rev-iff*[*reversal-rule*]: $rev\ u \in rev\ t* \longleftrightarrow u \in t*$
unfolding *root-def*[*reversed*] **using** *root-def*..

2.16 Commutation

The solution of the easiest nontrivial word equation, $x \cdot y = y \cdot x$, is in fact already contained in List.thy as the fact $xs \cdot ys = ys \cdot xs \Longrightarrow \exists\, m\ n\ zs$.

concat (*replicate m zs*) = *xs* ∧ *concat* (*replicate n zs*) = *ys*.

theorem *comm*: $x \cdot y = y \cdot x \longleftrightarrow (\exists\ t\ m\ k.\ x = t^{@}k \wedge y = t^{@}m)$
using *comm-append-are-replicate*[*of x y*, *folded pow-is-concat-replicate*] *pow-comm*
by *auto*

corollary *comm-root*: $x \cdot y = y \cdot x \longleftrightarrow (\exists\ t.\ x \in t* \wedge y \in t*)$
unfolding *root-def comm* **by** *fast*

lemma *commD*[*elim*]: $x \in t* \Longrightarrow y \in t* \Longrightarrow x \cdot y = y \cdot x$
using *comm-root* **by** *auto*

lemma *commE*[*elim*]: **assumes** $x \cdot y = y \cdot x$
obtains $t\ m\ k$ **where** $x = t^{@}k$ **and** $y = t^{@}m$
using *assms*[*unfolded comm*] **by** *blast*

lemma *comm-rootE*: **assumes** $x \cdot y = y \cdot x$
obtains t **where** $x \in t*$ **and** $y \in t*$
using *assms*[*unfolded comm-root*] **by** *blast*

2.17 Periods

Periodicity is probably the most studied property of words. It captures the fact that a word overlaps with itself. Another possible point of view is that the periodic word is a prefix of an (infinite) power of some nonempty word, which can be called its period word. Both these points of view are expressed by the following definition.

2.17.1 Periodic root

definition *period-root* :: *'a list* ⇒ *'a list* ⇒ *bool* (- ≤*p* -$^{\omega}$ [*51*,*51*] *60*)
where [*simp*]: *period-root u r* = (*u* ≤*p r* · *u* ∧ *r* ≠ *ε*)

lemma *per-rootI*[*simp*,*intro*]: $u \leq_p r \cdot u \Longrightarrow r \neq \varepsilon \Longrightarrow u \leq_p r^{\omega}$
by *simp*

lemma *per-rootI'*: **assumes** $u \leq_p r^{@}k$ **and** $r \neq \varepsilon$ **shows** $u \leq_p r^{\omega}$
using *per-rootI*[*OF pref-prod-pref*[*OF pref-pow-ext'*[*OF* ‹$u \leq_p r^{@}k$›] ‹$u \leq_p r^{@}k$›] ‹$r \neq \varepsilon$›].

lemma *per-rootD*[*elim*]: $u \leq_p r^{\omega} \Longrightarrow u \leq_p r \cdot u$
by *simp*

lemma *per-rootD'*: $u \leq_p r^{\omega} \Longrightarrow r \neq \varepsilon$
by *simp*

Empty word is not a periodic root but it has all nonempty periodic roots.

lemma *emp-any-per*: $r \neq \varepsilon \Longrightarrow (\varepsilon \leq_p r^{\omega})$

by *simp*

lemma *emp-not-per*: $\neg\ (x \leq_p \varepsilon^\omega)$
by *simp*

Any nonempty word is its own periodic root.

lemma *root-self*: $w \neq \varepsilon \Longrightarrow w \leq_p w^\omega$
by *simp*

"Short roots are prefixes"

lemma *root-is-take*: $w \leq_p r^\omega \Longrightarrow |r| \leq |w| \Longrightarrow\ r \leq_p w$
unfolding *period-root-def* **using** *pref-prod-long*[*of w r w*] **by** *fastforce*

Periodic words are prefixes of the power of the root, which motivates the notation

lemma *pref-pow-ext-take*: **assumes** $u \leq_p r^{@}k$ **shows** $u \leq_p take\ |r|\ u \cdot r^{@}k$
proof (*rule le-cases*[*of* $|u|$ $|r|$])
assume $|r| \leq |u|$
show $u \leq_p take\ |r|\ u \cdot r\ ^{@}\ k$
unfolding *pref-take*[*OF pref-prod-long*[*OF pref-pow-ext'*[*OF* ‹$u \leq_p r^{@}k$›] ‹$|r| \leq |u|$›]] **using** *pref-pow-ext'*[*OF* ‹$u \leq_p r^{@}k$›].
qed *simp*

lemma *pref-pow-take*: **assumes** $u \leq_p r^{@}k$ **shows** $u \leq_p take\ |r|\ u \cdot u$
using *pref-prod-pref*[*of u take* $|r|$ *u* $r^{@}k$, *OF pref-pow-ext-take* ‹$u \leq_p r^{@}k$›, *OF* ‹$u \leq_p r^{@}k$›].

lemma *per-exp*: **assumes** $u \leq_p r^\omega$ **shows** $u \leq_p r^{@}k \cdot u$
using *per-exp-pref*[*OF per-rootD*[*OF assms*]].

lemma *per-root-spref-powE*: **assumes** $u \leq_p r^\omega$
obtains k **where** $u <_p r^{@}k$
using *pref-prod-short'*[*OF per-exp*[*OF assms*, *of Suc* $|u|$] *long-power'*[*of r u*, *OF per-rootD'*[*OF assms*]]] **by** *blast*

lemma *period-rootE* [*elim*]: **assumes** $u \leq_p t^\omega$ **obtains** $n\ r$ **where** $r <_p t$ **and** $t^{@}n \cdot r = u$
proof–
obtain m **where** $u <_p t^{@}m$ **using** ‹$u \leq_p t^\omega$›
using *per-root-spref-powE* **by** *blast*
from *pref-mod-power*[*OF this that*, *of* $\lambda\ x\ y.\ y\ \lambda\ x\ y.\ x$]
show *?thesis* **by** *blast*
qed

lemma *per-add-exp*: **assumes** $u \leq_p r^\omega$ **and** $m \neq 0$ **shows** $u \leq_p (r^{@}m)^\omega$
using *per-exp-pref*[*OF per-rootD*, *OF* ‹$u \leq_p r^\omega$›, *of m*] *per-rootD'*[*OF* ‹$u \leq_p r^\omega$›, *folded nonzero-pow-emp*[*OF* ‹$m \neq 0$›, *of r*]]
unfolding *period-root-def*..

lemma *per-pref-ex*: **assumes** $u \leq p\ r^\omega$
obtains n **where** $u \leq p\ r^{@} n$ **and** $n \neq 0$
using *pcomp-shorter*[*OF ruler-pref*[*OF per-exp*[*OF* ‹$u \leq p\ r^\omega$›]]] *long-pow-ex*[*of r* $|u|$, *OF per-rootD'*[*OF* ‹$u \leq p\ r^\omega$›], *of thesis*]
by *blast*

theorem *per-pref*: $x \leq p\ r^\omega \longleftrightarrow (\exists\ k.\ x \leq p\ r^{@} k) \wedge r \neq \varepsilon$
using *per-pref-ex period-root-def pref-pow-ext' pref-prod-pref* **by** *metis*

lemma *pref-pow-conv*: $(\exists\ k.\ x \leq p\ r^{@} k) \longleftrightarrow (\exists\ k\ z.\ r^{@} k \cdot z = x \wedge z \leq p\ r)$
proof
assume $\exists k\ z.\ r\ ^{@}\ k \cdot z = x \wedge z \leq p\ r$
then obtain $k\ z$ **where** $r^{@} k \cdot z = x$ **and** $z \leq p\ r$ **by** *blast*
thus $\exists\ k.\ x \leq p\ r^{@} k$
using *pref-cancel'*[*OF* ‹$z \leq p\ r$›, *of* $r^{@} k$, *unfolded* ‹$r^{@} k \cdot z = x$›, *folded pow-Suc2-list*] **by** *fast*
next
assume $\exists\ k.\ x \leq p\ r^{@} k$ **then obtain** k **where** $x \leq p\ r^{@} k$ **by** *blast*
{assume $r = \varepsilon$
have $x = \varepsilon$
using *pref-emp*[*OF* ‹$x \leq p\ r\ ^{@}\ k$›[*unfolded* ‹$r = \varepsilon$› *emp-pow*]].
hence $\exists\ k\ z.\ r^{@} k \cdot z = x \wedge z \leq p\ r$
using ‹$r = \varepsilon$› *emp-pow* **by** *auto*}
moreover
{assume $r \neq \varepsilon$ **have** $x <p\ r^{@}(Suc\ k)$
using *pref-ext-nemp*[*OF* ‹$x \leq p\ r^{@} k$› ‹$r \neq \varepsilon$›, *folded pow-Suc2-list*].
then have $\exists\ k\ z.\ r^{@} k \cdot z = x \wedge z \leq p\ r$
using *pref-mod-power*[*OF pref-ext-nemp*[*OF* ‹$x \leq p\ r^{@} k$› ‹$r \neq \varepsilon$›, *folded pow-Suc2-list*], *of* $\exists\ k\ z.\ r^{@} k \cdot z = x \wedge z \leq p\ r$]
by *auto*}
ultimately show $\exists\ k\ z.\ r^{@} k \cdot z = x \wedge z \leq p\ r$ **by** *blast*
qed

lemma *per-eq*: $x \leq p\ r^\omega \longleftrightarrow (\exists\ k\ z.\ r^{@} k \cdot z = x \wedge z \leq p\ r) \wedge r \neq \varepsilon$
using *per-pref*[*unfolded pref-pow-conv*].

The previous theorem allows to prove some basic relations between powers, periods and commutation

lemma *per-drop-exp*: $u \leq p\ (r^{@} m)^\omega \implies u \leq p\ r^\omega$
using *per-pref*[*of u* $r^{@} m$] *pow-mult-list*[*of r m*, *symmetric*] **unfolding** *per-pref*[*of u r*]
using *nemp-pow'*[*of r m*] **by** *auto*

lemma *per-drop-exp'*: $i \neq 0 \implies p \leq p\ e^{@} i \cdot p \implies p \leq p\ e \cdot p$
by (*metis period-root-def eq-pow-exp per-drop-exp pow-zero*)

lemma *per-drop-exp-rev*: **assumes** $i \neq 0\ p \leq s\ p \cdot e^{@} i$ **shows** $p \leq s\ p \cdot e$
using *per-drop-exp'*[*OF* ‹$i \neq 0$› ‹$p \leq s\ p \cdot e^{@} i$›[*unfolded suffix-to-prefix rev-append rev-pow*], *folded rev-append suffix-to-prefix*].

corollary *comm-drop-exp*: **assumes** $m \neq 0$ **and** $u \cdot r^{@}m = r^{@}m \cdot u$ **shows** $r \cdot u = u \cdot r$
proof−
  {assume $r \neq \varepsilon$ $u \neq \varepsilon$
    hence $u{\cdot}r \leq p$ $u{\cdot}r^{@}m$ **using** *pop-pow-one* ‹$m \neq 0$›
      by (*simp add*: *pop-pow-one*)
    have $u{\cdot}r \leq p$ $r^{@}m{\cdot}(u{\cdot}r)$
     using *pref-ext*[*of* $u \cdot r$ $r^{@}m \cdot u$ r, *unfolded append-assoc*[*of* $r^{@}m$ u r], *OF* ‹$u{\cdot}r \leq p$ $u{\cdot}r^{@}m$›[*unfolded* ‹$u \cdot r^{@}m = r^{@}m \cdot u$›]]**.**
    hence $u{\cdot}r \leq p$ $r{\cdot}(u{\cdot}r)$
      using *per-drop-exp*[*of* $u{\cdot}r$ r m] ‹$m \neq 0$› *per-drop-exp′* **by** *blast*
    from *comm-ruler*[*OF self-pref*[*of* $r \cdot u$], *of* $r \cdot u \cdot r$, *OF this*]
    have $r \cdot u = u \cdot r$ **by** *auto*
  }
  thus $r \cdot u = u \cdot r$
    by *fastforce*
qed

corollary *pow-comm-comm*: **assumes** $x^{@}j = y^{@}k$ **and** $j \neq 0$ **shows** $x{\cdot}y = y{\cdot}x$
proof(*cases* $k = 0$)
  assume $k = 0$
 from ‹$x^{@}j = y^{@}k$›[*unfolded zero-exp*[*OF this*, *of y*], *unfolded nonzero-pow-emp*[*OF* ‹$j \neq 0$›]]
  show $x \cdot y = y \cdot x$ **by** *simp*
next
  assume $k \neq 0$ **show** $x \cdot y = y \cdot x$
    using *comm-drop-exp*[*of j y x*, *OF* ‹$j \neq 0$›, *OF*
        comm-drop-exp[*of k* $x^{@}j$ *y*, *OF* ‹$k \neq 0$› , *unfolded* ‹$x^{@}j = y^{@}k$›, *OF refl*, *folded* ‹$x^{@}j = y^{@}k$›]]**.**
qed

corollary *comm-pow-roots*: **assumes** $m \neq 0$ $k \neq 0$
  shows $u^{@}m \cdot v^{@}k = v^{@}k \cdot u^{@}m \longleftrightarrow u \cdot v = v \cdot u$
  using *comm-drop-exp*[*OF* ‹$k \neq 0$›, *of* $u^{@}m$ *v*, *THEN comm-drop-exp*[*OF* ‹$m \neq 0$›, *of v u*]]
        comm-add-exps[*of u v m k*] **by** *blast*

lemma *pow-comm-comm′*: **assumes** *comm*: $u^{@}(Suc\ k) = v^{@}(Suc\ l)$ **shows** $u \cdot v = v \cdot u$
  using *comm pow-comm-comm* **by** *blast*

lemma *comm-trans*: **assumes** *uv*: $u{\cdot}v = v{\cdot}u$ **and** *vw*: $w{\cdot}v = v{\cdot}w$ **and** *nemp*: $v \neq \varepsilon$ **shows** $u \cdot w = w \cdot u$
proof −
  consider (*u-emp*) $u = \varepsilon$ | (*w-emp*) $w = \varepsilon$ | (*nemp′*) $u \neq \varepsilon$ **and** $w \neq \varepsilon$ **by** *blast*
  then show *?thesis* **proof** (*cases*)
    case *nemp′*
      have *eq*: $u^{@}(|v| * |w|) = w^{@}(|v| * |u|)$

unfolding *power-mult comm-common-power*[*OF uv*] *comm-common-power*[*OF vw*]
unfolding *pow-mult-list*[*symmetric*] *mult.commute*[*of* |*u*|]..
obtain *k l* **where** *k*: |*v*| ∗ |*w*| = *Suc k* **and** *l*: |*v*| ∗ |*u*| = *Suc l*
using *nemp nemp′* **unfolding** *length-0-conv*[*symmetric*]
using *not0-implies-Suc*[*OF no-zero-divisors*]
by *presburger*
show *?thesis*
using *pow-comm-comm′*[*OF eq*[*unfolded k l*]].
qed *simp*+
qed

theorem *per-all-exps*: ⟦ *m* ≠ *0*; *k* ≠ *0* ⟧ ⟹ (*u* ≤*p* (*r*^@*m*)^ω) ⟷ (*u* ≤*p* (*r*^@*k*)^ω)
using *per-drop-exp*[*of u r m*, *THEN per-add-exp*[*of u r k*]] *per-drop-exp*[*of u r k*, *THEN per-add-exp*[*of u r m*]] **by** *blast*

lemma *drop-per-pref*: **assumes** *w* ≤*p* *u*^ω **shows** *drop* |*u*| *w* ≤*p* *w*
using *pref-drop*[*OF per-rootD*[*OF* ‹*w* ≤*p* *u*^ω›], *of* |*u*|, *unfolded drop-pref*[*of u w*]].

lemma *per-root-trans*[*trans*]: *w* ≤*p* *u*^ω ⟹ *u* ∈ *t*∗ ⟹ *w* ≤*p* *t*^ω
using *root-def*[*of u t*] *per-drop-exp*[*of w t*] **by** *blast*

Note that ⟦*w* ≤*p* *u*^ω; *u* ≤*p* *t*^ω⟧ ⟹ *w* ≤*p* *t*^ω does not hold.

lemma *per-root-same-prefix*:*w* ≤*p* *r*^ω ⟹ *w′* ≤*p* *r*^ω ⟹ *w* ⋈ *w*
by *blast*

lemma *take-after-drop*: |*u*| + *q* ≤ |*w*| ⟹ *w* ≤*p* *u*^ω ⟹ *take q* (*drop* |*u*| *w*) = *take q w*
using *pref-share-take*[*OF drop-per-pref*[*of w u*] *len-after-drop*[*of* |*u*| *q w*]].

The following lemmas are a weak version of the Periodicity lemma

lemma *two-pers′*:
assumes *pu*: *w* ≤*p* *u* · *w* **and** *pv*: *w* ≤*p* *v* · *w* **and** *len*: |*u*| + |*v*| ≤ |*w*|
shows *u* · *v* = *v* · *u*
proof−
have *uv*: *w* ≤*p* (*u* · *v*) · *w* **using** *pref-prolong*[*OF pu pv*] **unfolding** *lassoc*.
have *vu*: *w* ≤*p* (*v* · *u*) · *w* **using** *pref-prolong*[*OF pv pu*] **unfolding** *lassoc*.
have *u* · *v* ≤*p* *w* **using** *len pref-prod-long*[*OF uv*] **by** *simp*
moreover have *v* · *u* ≤*p* *w* **using** *len pref-prod-long*[*OF vu*] **by** *simp*
ultimately show *u* · *v* = *v* · *u* **by** (*rule pref-comp-eq*[*unfolded prefix-comparable-def*, *OF ruler swap-len*])
qed

lemma *two-pers*: **assumes** *w* ≤*p* *u*^ω **and** *w* ≤*p* *v*^ω **and** |*u*|+|*v*| ≤ |*w*| **shows** *u*·*v* = *v*·*u*
using *two-pers′*[*OF per-rootD*[*OF assms*(*1*)] *per-rootD*[*OF assms*(*2*)] *assms*(*3*)].

2.17.2 Period - numeric

Definition of a period as the length of the periodic root is often offered as the basic one. From our point of view, it is secondary, and less convenient for reasoning.

definition *periodN* :: *'a list* ⇒ *nat* ⇒ *bool*
  where [*simp*]: *periodN w n* = $w \leq p\ (take\ n\ w)^\omega$

lemma *periodN-I*: $w \neq \varepsilon \Longrightarrow n \neq 0 \Longrightarrow w \leq p\ (take\ n\ w) \cdot w \Longrightarrow periodN\ w\ n$
  unfolding *periodN-def period-root-def* **by** *fastforce*

The numeric definition respects the following convention about empty words and empty periods.

lemma *emp-no-periodN*: ¬ *periodN* ε *n*
  by *simp*

lemma *zero-not-per*: ¬ *periodN w 0*
  by *simp*

lemma *periodN-D1*: *periodN w n* ⟹ $w \neq \varepsilon$
  by *simp*

lemma *periodN-D2*: *periodN w n* ⟹ $n \neq 0$
  by *simp*

lemma *periodN-D3*: *periodN w n* ⟹ $w \leq p\ take\ n\ w \cdot w$
  by *simp*

A nonempty word has all "long" periods

lemma *all-long-pers*: ⟦ $w \neq \varepsilon$; $|w| \leq n$ ⟧ ⟹ *periodN w n*
  by *simp*

lemmas *per-nemp* = *periodN-D1*

lemmas *per-positive* = *periodN-D2*

The standard numeric definition of a period uses indeces.

lemma *periodN-indeces*: **assumes** *periodN w n* **and** $i + n < |w|$ **shows** $w!i = w!(i+n)$
proof−
  have $w\ !\ i = (take\ n\ w \cdot w)\ !\ (n + i)$

```
    using nth-append-length-plus[of take n w w i, symmetric]
    unfolding take-len[OF less-imp-le[OF add-lessD2[OF ‹i + n < |w|›]]].
  also have ... = w ! (i + n)
    using pref-index[OF periodN-D3[OF ‹periodN w n›] ‹i + n < |w|›, symmetric]
unfolding add.commute[of n i].
  finally show ?thesis.
qed

lemma indeces-periodN:
  assumes  w ≠ ε and n ≠ 0 and  forall: ⋀ i. i + n < |w| ⟹ w!i = w!(i+n)
  shows periodN w n
proof–
  have |w| ≤ |take n w · w|
    by auto
  {fix j assume j < |w|
    have w ! j = (take n w · w) ! j
    proof (cases j < |take n w|)
      assume j < |take n w| show w ! j = (take n w · w) ! j
        using pref-index[OF take-is-prefix ‹j < |take n w|›, symmetric]
        unfolding pref-index[OF triv-pref ‹j < |take n w|›, of w].
    next
      assume ¬ j < |take n w|
      from leI[OF this] ‹j < |w|›
      have |take n w| = n
        by force
      hence j = (j − n) + n and (j − n) + n < |w|
        using  leI[OF ‹¬ j < |take n w|›] ‹j < |w|› by simp+
      hence w!j = w!(j − n)
        using forall by simp
      from this[folded nth-append-length-plus[of take n w w j−n, unfolded ‹|take n
w| = n›]]
      show w ! j = (take n w · w) ! j
        using ‹j = (j − n) + n› by simp
    qed}
  with index-pref[OF ‹|w| ≤ |take n w · w|›]
  have w ≤p take n w · w by blast
  thus ?thesis
    using assms by force
qed
```

In some cases, the numeric definition is more useful than the definition using the period root.

```
lemma periodN-rev: assumes periodN w p shows periodN (rev w) p
proof (rule indeces-periodN[of rev w p, OF - periodN-D2[OF assms]])
  show rev w ≠ ε
    using assms[unfolded periodN-def period-root-def] by force
next
  fix i assume i + p < |rev w|
  from this[unfolded length-rev] add-lessD1
```

have $i < |w|$ **and** $i + p < |w|$ **by** *blast+*
have *e*: $|w| - Suc\ (i + p) + p = |w| - Suc\ i$ **using** ‹$i + p < |rev\ w|$› **by** *simp*
have $|w| - Suc\ (i + p) + p < |w|$ **using** ‹$i + p < |w|$› **by** *linarith*
from *periodN-indeces*[*OF assms this*] *rev-nth*[*OF* ‹$i < |w|$›, *folded e*] *rev-nth*[*OF* ‹$i + p < |w|$›]
show *rev w* ! i = *rev w* !(i+p) **by** *presburger*
qed

lemma *periodN-rev-conv* [*reversal-rule*]: *periodN* (*rev w*) $n \longleftrightarrow$ *periodN w n*
using *periodN-rev periodN-rev*[*of rev w*] **unfolding** *rev-rev-ident* **by** (*intro iffI*)

lemma *periodN-fac*: **assumes** *periodN* (u·w·v) p **and** $w \neq \varepsilon$
shows *periodN w p*
proof (*rule indeces-periodN*, *simp add*: ‹$w \neq \varepsilon$›)
show $p \neq 0$ **using** *periodN-D2*[*OF* ‹*periodN* (u·w·v) p›].
fix i **assume** $i + p < |w|$
hence $|u| + i + p < |u·w·v|$
by *simp*
from *periodN-indeces*[*OF* ‹*periodN* (u·w·v) p› *this*]
have (u·w·v)!($|u| + i$) = (u·w·v)! ($|u| + (i + p)$)
by (*simp add*: *add.assoc*)
thus $w!i = w!(i+p)$
using *nth-append-length-plus*[*of u w*·*v* , *unfolded lassoc*] ‹$i + p < |w|$› *add-lessD1*[*OF* ‹$i + p < |w|$›]
nth-append[*of w v*] **by** *auto*
qed

lemma *periodN-fac′*: *periodN v p* $\Longrightarrow$ $u \leq_f v$ $\Longrightarrow$ $u \neq \varepsilon$ $\Longrightarrow$ *periodN u p*
by (*elim facE*, *hypsubst*, *rule periodN-fac*)

lemma assumes $y \neq \varepsilon$ **and** $k \neq 0$ **shows** $y^{@}k \neq \varepsilon$
by (*simp add*: *assms*(*1*) *assms*(*2*) *nemp-emp-power*)

lemma *pow-per*: **assumes** $y \neq \varepsilon$ **and** $k \neq 0$ **shows** *periodN* ($y^{@}k$) $|y|$
using *periodN-I*[*OF* ‹$k \neq 0$›[*folded nemp-emp-power*[*OF* ‹$y \neq \varepsilon$›]] *nemp-len*[*OF* ‹$y \neq \varepsilon$›]] *pref-pow-ext-take* **by** *blast*

lemma *per-fac*: **assumes** $y \neq \varepsilon$ **and** $v \neq \varepsilon$ **and** $y^{@}k = u·v·w$ **shows** *periodN v* $|y|$
proof−
have $k \neq 0$
using *nemp-pow suf-nemp*[*OF pref-nemp*[*OF* ‹$v \neq \varepsilon$›, *of w*], *of u*, *folded* ‹$y^{@}k = u·v·w$›] **by** *blast*
from *pow-per*[*OF* ‹$y \neq \varepsilon$› *this*, *unfolded* ‹$y^{@}k = u·v·w$›]
have *periodN* ($u \cdot v \cdot w$) $|y|$.
from *periodN-fac*[*OF this* ‹$v \neq \varepsilon$›]
show *periodN v* $|y|$.

qed

The numeric definition is equivalent to being prefix of a power.

theorem *periodN-pref*: *periodN w n* $\longleftrightarrow$ ($\exists$ *k r*. *w* $\leq$*np* *r*$^{@}$*k* $\wedge$ |*r*| = *n*) (**is** - $\longleftrightarrow$ *?R*)
proof(*cases w* = ε, *simp*)
assume *w* $\neq$ ε
show *periodN w n* $\longleftrightarrow$ *?R*
proof
assume *periodN w n*
consider (*short*) |*w*| $\leq$ *n* | (*long*) *n* < |*w*|
by *linarith*
then show *?R*
proof(*cases*)
assume |*w*| $\leq$ *n*
from *le-add-diff-inverse*[*OF this*]
obtain *z* **where** |*w* · *z*| = *n*
unfolding *length-append* **using** *exE*[*OF Ex-list-of-length*[*of n* − |*w*|]] **by** *metis*
thus *?R*
using *pow-one-id npI'*[*OF* ‹*w* $\neq$ ε›] **by** *metis*
next
assume *n* < |*w*|
then show *?R*
using ‹*periodN w n*›[*unfolded periodN-def per-pref*[*of w take n w*]]
‹*w* $\neq$ ε› *take-len*[*OF less-imp-le*[*OF* ‹*n* < |*w*|›]] **by** *blast*
qed
next
assume *?R*
then obtain *k r* **where** *w* $\leq$*np* *r*$^{@}$*k* **and** *n* = |*r*| **by** *blast*
have *w* $\leq$*p* *take n w* · *w*
using *pref-pow-take*[*OF npD*[*OF* ‹*w* $\leq$*np* *r* $^{@}$ *k*›], *folded* ‹*n* = |*r*|›].
have *n* $\neq$ *0*
unfolding *length-0-conv*[*of r*, *folded* ‹*n* = |*r*|›] **using** ‹*w* $\leq$*np* *r* $^{@}$ *k*› **by** *force*
hence *take n w* $\neq$ ε
unfolding ‹*n* = |*r*|› **using** ‹*w* $\neq$ ε› **by** *simp*
thus *periodN w n*
unfolding *periodN-def period-root-def* **using** ‹*w* $\leq$*p* *take n w* · *w*› **by** *blast*
qed
qed

Two more characterizations of a period

theorem *per-shift*: **assumes** *w* $\neq$ ε *n* $\neq$ *0*
shows *periodN w n* $\longleftrightarrow$ *drop n w* $\leq$*p* *w*
proof
assume *periodN w n* **show** *drop n w* $\leq$*p* *w*
using *drop-per-pref*[*OF* ‹*periodN w n*›[*unfolded periodN-def*]]
append-take-drop-id[*of n w*, *unfolded append-eq-conv-conj*] **by** *argo*
next

assume *drop n w* $\leq_p$ *w*
show *periodN w n*
using *conjI*[*OF pref-cancel'*[*OF* ‹*drop n w* $\leq_p$ *w*›, *of take n w*] *take-nemp*[*OF* ‹$w \neq \varepsilon$› ‹$n \neq 0$›]]
unfolding *append-take-drop-id period-root-def* **by** *force*
qed

lemma *rotate-per-root*: **assumes** $w \neq \varepsilon$ **and** $n \neq 0$ **and** $w = rotate\ n\ w$
shows *periodN w n*
proof (*cases* $|w| \leq n$)
assume $|w| \leq n$
from *all-long-pers*[*OF* ‹$w \neq \varepsilon$›, *OF this*]
show *periodN w n*.
next
assume *not*: $\neg\ |w| \leq n$
have *drop* (n *mod* $|w|$) *w* $\leq_p$ *w*
using *prefI*[*OF rotate-drop-take*[*symmetric*, *of n w*]]
unfolding ‹$w = rotate\ n\ w$›[*symmetric*].
from *per-shift*[*OF* ‹$w \neq \varepsilon$› ‹$n \neq 0$›] *this*[*unfolded mod-less*[*OF not*[*unfolded not-le*]]]
show *periodN w n*..
qed

Various lemmas on periods

lemma *periodN-drop*: **assumes** *periodN w p* **and** $p < |w|$
shows *periodN* (*drop p w*) *p*
using *periodN-fac*[*of take p w drop p w* ε *p*] ‹$p < |w|$› ‹*periodN w p*›
unfolding *append-take-drop-id drop-eq-Nil not-le append-Nil2* **by** *blast*

lemma *ext-per-left*: **assumes** *periodN w p* **and** $p \leq |w|$
shows *periodN* (*take p w* · *w*) *p*
proof–
have *f*: *take p* (*take p w* · *w*) = *take p w*
using ‹$p \leq |w|$› **by** *simp*
show *?thesis*
using ‹*periodN w p*› *pref-cancel'*[*of w take p w* · *w take p w*] **unfolding** *f periodN-def period-root-def*
by *blast*
qed

lemma *ext-per-left-power*: *periodN w p* $\Longrightarrow$ $p \leq |w|$ $\Longrightarrow$ *periodN* ((*take p w*)$^{@}k$ · *w*) *p*
proof (*induction k*)
case (*Suc k*)
show *?case*
using *ext-per-left*[*OF Suc.IH*[*OF* ‹*periodN w p*› ‹$p \leq |w|$›]] ‹$p \leq |w|$›
unfolding *pref-share-take*[*OF per-exp-pref*[*OF periodN-D3*[*OF* ‹*periodN w p*›]] ‹$p \leq |w|$›,*symmetric*]

lassoc pow-Suc-list[*symmetric*] **by** *fastforce*
qed *auto*

lemma *take-several-pers*: **assumes** *periodN w n* **and** $m*n \leq |w|$
shows $(take\ n\ w)^{@}m = take\ (m*n)\ w$
proof (*cases* $m = 0$, *simp*)
assume $m \neq 0$
have $|(take\ n\ w)^{@}m| = m*n$
unfolding *pow-len nat-prod-le*[*OF* ‹$m \neq 0$› ‹$m*n \leq |w|$›, *THEN take-len*] **by** *blast*
have $(take\ n\ w)^{@}m \leq_p w$
using ‹*periodN w n*›[*unfolded periodN-def*, *THEN per-exp*[*of w take n w m*], *THEN*
ruler-le[*of take n* $w^{@}m$ *take n* $w^{@}m \cdot w\ w$, *OF triv-pref*], *OF* ‹$m * n \leq |w|$›[*folded* ‹$|take\ n\ w^{@}m| = m * n$›]].
show *?thesis*
using *pref-take*[*OF* ‹$take\ n\ w^{@}m \leq_p w$›, *unfolded* ‹$|take\ n\ w^{@}m| = m * n$›, *symmetric*].
qed

lemma *per-mult*: **assumes** *periodN w n* **and** $m \neq 0$ **shows** *periodN w* $(m*n)$
proof (*cases* $m*n \leq |w|$)
have $w \neq \varepsilon$ **using** *periodN-D1*[*OF* ‹*periodN w n*›].
assume $\neg\ m * n \leq |w|$ **thus** *periodN w* $(m*n)$
using *all-long-pers*[*of* $w\ m * n$, *OF* ‹$w \neq \varepsilon$›] **by** *linarith*
next
assume $m * n \leq |w|$
show *periodN w* $(m*n)$
using *per-add-exp*[*of w take n w*, *OF - ‹m ≠ 0›*] ‹*periodN w n*›
unfolding *periodN-def period-root-def take-several-pers*[*OF* ‹*periodN w n*› ‹$m*n \leq |w|$›, *symmetric*] **by** *blast*
qed

lemma *root-periodN*: **assumes** $w \neq \varepsilon$ **and** $w \leq_p r^{\omega}$ **shows** *periodN w* $|r|$
unfolding *periodN-def period-root-def* **using** *per-pref-ex*[*OF* ‹$w \leq_p r^{\omega}$›
pref-pow-take[*of w r*], *of* $\lambda\ x.\ x$] *take-nemp-len*[*OF* ‹$w \neq \varepsilon$› *per-rootD'*[*OF* ‹$w \leq_p r^{\omega}$›]] **by** *blast*

theorem *two-periodsN*:
assumes *periodN w p periodN w q* $p + q \leq |w|$
shows *periodN w* $(gcd\ p\ q)$
proof−
obtain t **where** $take\ p\ w \in t*\ take\ q\ w \in t*$
using *two-pers*[*OF* ‹*periodN w p*›[*unfolded periodN-def*] ‹*periodN w q*›[*unfolded periodN-def*],
unfolded take-len[*OF add-leD1*[*OF* ‹$p + q \leq |w|$›]] *take-len*[*OF add-leD2*[*OF* ‹$p + q \leq |w|$›]],
OF ‹$p + q \leq |w|$›, *unfolded comm-root*[*of take p w take q w*]] **by** *blast*
hence $w \leq_p t^{\omega}$

using ‹*periodN w p*› *periodN-def per-root-trans* **by** *blast*
have *periodN w* |*t*|
using *root-periodN*[*OF per-nemp*[*OF* ‹*periodN w p*›] ‹*w* ≤*p t*^*ω*›].
have |*t*| *dvd* (*gcd p q*)
using *gcd-nat.boundedI*[*OF root-len-dvd*[*OF* ‹*take p w* ∈ *t**›] *root-len-dvd*[*OF* ‹*take q w* ∈ *t**›]]
unfolding *take-len*[*OF add-leD1*[*OF* ‹*p* + *q* ≤ |*w*|›]] *take-len*[*OF add-leD2*[*OF* ‹*p* + *q* ≤ |*w*|›]].
thus *?thesis*
using *per-mult*[*OF* ‹*periodN w* |*t*|›, *of gcd p q div* |*t*|, *unfolded dvd-div-mult-self*[*OF* ‹|*t*| *dvd* (*gcd p q*)›]]
dvd-div-mult-self[*OF* ‹|*t*| *dvd* (*gcd p q*)›]
gcd-eq-0-iff[*of p q*] *mult-zero-left*[*of* |*t*|] *periodN-D2*[*OF* ‹*periodN w p*›] **by** *argo*
qed

lemma *index-mod-per-root*: **assumes** *r* ≠ *ε* **and** *i*: ∀ *i* < |*w*|. *w*!*i* = *r*!(*i mod* |*r*|)
shows *w* ≤*p r*^*ω*
proof–
have *i* < |*w*| ⟹ (*r* · *w*) ! *i* = *r* ! (*i mod* |*r*|) **for** *i*
by (*simp add*: *i mod-if nth-append*)
hence *w* ≤*p r* · *w*
using *index-pref*[*of w r* · *w*] *i*
by *simp*
thus *?thesis* **unfolding** *period-root-def* **using** ‹*r* ≠ *ε*› **by** *auto*
qed

lemma *index-pref-pow-mod*: *w* ≤*p r*^@*k* ⟹ *i* < |*w*| ⟹ *w*!*i* = *r*!(*i mod* |*r*|)
using *index-pow-mod*[*of i r k*] *less-le-trans*[*of i* |*w*| |*r*^@*k*|] *prefix-length-le*[*of w r*^@*k*] *pref-index*[*of w r*^@*k i*] **by** *argo*

lemma *index-per-root-mod*: *w* ≤*p r*^*ω* ⟹ *i* < |*w*| ⟹ *w*!*i* = *r*!(*i mod* |*r*|)
using *index-pref-pow-mod*[*of w r - i*] *per-pref*[*of w r*] **by** *blast*

lemma *root-divisor*: **assumes** *periodN w k* **and** *k dvd* |*w*| **shows** *w* ∈ (*take k w*)*
using *rootI*[*of take k w* (|*w*| *div k*)] **unfolding**
take-several-pers[*OF* ‹*periodN w k*›, *of* |*w*| *div k*, *unfolded dvd-div-mult-self*[*OF* ‹*k dvd* |*w*|›] *take-self*, *OF* , *OF order-refl*].

lemma *per-pref'*: **assumes** *u* ≠ *ε* **and** *periodN v k* **and** *u* ≤*p v* **shows** *periodN u k*
proof–
{ assume *k* ≤ |*u*|
have *take k v* = *take k u*
using *pref-share-take*[*OF* ‹*u* ≤*p v*› ‹*k* ≤ |*u*|›] **by** *auto*
hence *take k v* ≠ *ε*
using ‹*periodN v k*› **by** *auto*
hence *take k u* ≠ *ε*
by (*simp add*: ‹*take k v* = *take k u*›)
have *u* ≤*p take k u* · *v*

```
    using ‹periodN v k›
    unfolding periodN-def period-root-def ‹take k v = take k u›
    using pref-trans[OF ‹u ≤p v›, of take k u · v]
    by blast
  hence u ≤p take k u · u
    using ‹u ≤p v› pref-prod-pref by blast
  hence ?thesis
    using ‹take k u ≠ ε› periodN-def by fast
 }
 thus ?thesis
  using ‹u ≠ ε› all-long-pers nat-le-linear by blast
qed
```

2.17.3 Period: overview

```
notepad
begin
 fix w r::'a list
 fix n::nat
 assume w ≠ ε r ≠ ε n > 0
 have ¬ w ≤p ε^ω
  by simp
 have ¬ ε ≤p ε^ω
  by simp
 have ε ≤p r^ω
  by (simp add: ‹r ≠ ε›)
 have ¬ periodN w 0
  by simp
 have ¬ periodN ε 0
  by simp
 have ¬ periodN ε n
  by simp
end
```

2.17.4 Singleton and its power

```
lemma concat-len-one: assumes |us| = 1 shows concat us = hd us
 using concat-sing[OF sing-word[OF ‹|us| = 1›, symmetric]].

lemma sing-pow-hd-tl: c # w ∈ [a]* ⟷ c = a ∧ w ∈ [a]*
proof
 assume c = a ∧ w ∈ [a]*
 thus c # w ∈ [a]*
  unfolding hd-word[of - w] using add-root[of [c] w] by simp
next
 assume c # w ∈ [a]*
 then obtain k where [a]@k = c # w unfolding root-def by blast
 thus c = a ∧ w ∈ [a]*
 proof (cases k = 0, simp)
  assume [a] @ k = c # w and k ≠ 0
```

from *eqd-equal*[*of* [*a*], *OF this*(*1*)[*unfolded hd-word*[*of* - *w*] *pop-pow-one*[*OF* ‹*k* ≠ *0*›]]]
show *?thesis*
unfolding *root-def* **by** *auto*
qed
qed

lemma *pref-sing-power*: **assumes** $w \leq p\ [a]^{@}m$ **shows** $w = [a]^{@}(|w|)$
proof−
have $[a]^{@}m = [a]^{@}(|w|)\cdot[a]^{@}(m-|w|)$
using *pop-pow*[*OF prefix-length-le*[*OF assms*, *unfolded sing-pow-len*], *of* [*a*], *symmetric*].
show *?thesis*
using *conjunct1*[*OF eqd-equal*[*of w* $w^{-1>}$[*a*]$^{@}m$ [*a*]$^{@}(|w|)$[*a*]$^{@}(m-|w|)$,
unfolded lq-pref[*OF assms*] *sing-pow-len*,
OF ‹[*a*]$^{@}m$ = [*a*]$^{@}(|w|)\cdot$[*a*]$^{@}(m-|w|)$› *refl*]].
qed

lemma *sing-pow-palindrom*: **assumes** $w = [a]^{@}k$ **shows** *rev w* = *w*
using *rev-pow*[*of* [*a*] |*w*|, *unfolded rev-sing*]
unfolding *pref-sing-power*[*of w a k*, *unfolded assms*[*unfolded root-def*, *symmetric*], *OF self-pref*, *symmetric*].

lemma *suf-sing-power*: **assumes** $w \leq s\ [a]^{@}m$ **shows** $w \in [a]*$
using *sing-pow-palindrom*[*of rev w a* |*rev w*|, *unfolded rev-rev-ident*]
pref-sing-power[*of rev w a m*, *OF* ‹*w* ≤*s* [*a*]$^{@}$*m*›[*unfolded suffix-to-prefix rev-pow rev-rev-ident rev-sing*]]
rootI[*of* [*a*] |*rev w*|] **by** *auto*

lemma *sing-fac-pow*: **assumes** $w \in [a]*$ **and** $v \leq f\ w$ **shows** $v \in [a]*$
proof−
obtain *k* **where** $w = [a]^{@}k$ **using** ‹*w* ∈ [*a*]∗›[*unfolded root-def*] **by** *blast*
obtain *p* **where** $p \leq p\ w$ **and** $v \leq s\ p$
using *fac-pref-suf*[*OF* ‹ $v \leq f\ w$›] **by** *blast*
hence $v \leq s\ [a]^{@}\ |p|$
using *pref-sing-power*[*OF* ‹$p \leq p\ w$›[*unfolded* ‹*w* = [*a*]$^{@}$*k*›]] **by** *argo*
from *suf-sing-power*[*OF this*]
show *?thesis*.
qed

lemma *sing-pow-fac′*: **assumes** $a \neq b$ **and** $w \in [a]*$ **shows** ¬ ([*b*] ≤*f w*)
using *sing-fac-pow*[*OF* ‹ *w* ∈ [*a*]∗›, *of* [*b*]] **unfolding** *sing-pow-hd-tl*[*of b* ε]
using ‹*a* ≠ *b*› **by** *auto*

lemma *all-set-sing-pow*: (∀ *b*. *b* ∈ *set w* ⟶ *b* = *a*) ⟷ *w* ∈ [*a*]∗ (**is** *?All* ⟷ -)
proof
assume *?All*
then show *w* ∈ [*a*]∗
proof (*induct w*, *simp*)

```
    case (Cons c w)
    then show ?case
      by (simp add: sing-pow-hd-tl)
  qed
  next
  assume w ∈ [a]∗
  then show ?All
  proof (induct w, simp)
    case (Cons c w)
    then show ?case
      unfolding sing-pow-hd-tl by simp
  qed
qed

lemma sing-pow-set: u ∈ [a]∗ ⟹ u ≠ ε ⟹ set u = {a}
  unfolding all-set-sing-pow[symmetric]
  using hd-in-set[of u] is-singletonI'[unfolded is-singleton-the-elem, of set u]
    singleton-iff[of a the-elem (set u)]
  by auto

lemma takeWhile-sing-root: takeWhile (λ x. x = a) w ∈ [a]∗
  unfolding all-set-sing-pow[symmetric] using set-takeWhileD[of - λ x. x = a w]
by blast

lemma takeWhile-sing-pow: takeWhile (λ x. x = a) w = w ⟷ w = [a]^@|w|
  by(induct w,  auto)

lemma dropWhile-sing-pow: dropWhile (λ x. x = a) w = ε ⟷ w = [a]^@|w|
  by(induct w,  auto)

lemma distinct-letter-in: assumes ¬ w ∈ [a]∗
  obtains m b q where [a]^@m · [b] · q = w and b ≠ a
proof−
  have dropWhile (λ x. x = a) w ≠ ε
    unfolding dropWhile-sing-pow using assms rootI[of [a] |w|] by auto
   hence eq: takeWhile (λ x. x = a) w · [hd (dropWhile (λ x. x = a) w)] · tl
(dropWhile (λ x. x = a) w) = w
    by simp
  have root:takeWhile (λ x. x = a) w ∈ [a]∗
    by (simp add: takeWhile-sing-root)
  have  hd (dropWhile (λ x. x = a) w) ≠ a
    using hd-dropWhile[OF ‹dropWhile (λx. x = a) w ≠ ε›].
  from that[OF -  this]
  show thesis
    using eq root unfolding root-def by metis
qed

lemma distinct-letter-in-hd: assumes ¬ w ∈  [hd w]∗
  obtains m b q where  [hd w]^@m · [b] · q = w and b ≠ hd w and m ≠ 0
```

proof–
obtain $m\ b\ q$ **where** *a1*: $[hd\ w]^{@}m \cdot [b] \cdot q = w$ **and** *a2*: $b \neq hd\ w$
using *distinct-letter-in*[*OF assms*]**.**
hence $m \neq 0$
using *power-eq-if*[*of* $[hd\ w]$ m] *list.sel*(*1*) **by** *fastforce*
from *that a1 a2 this*
show *thesis* **by** *blast*
qed

lemma *distinct-letter-in-suf*: **assumes** $\neg\ w \in [a]*$ **shows** $\exists\ m\ b.\ [b] \cdot [a]^{@}m \leq s\ w \wedge b \neq a$
proof–
have $\neg\ rev\ w \in [a]*$
using *rev-pow*[*of* $[a]$] **unfolding** *rev-sing* **using** *assms root-def rev-rev-ident*[*of* w]
by *metis*
obtain $m\ b\ q$ **where** $[a]^{@}m \cdot [b] \cdot q = rev\ w$ **and** $b \neq a$
using *distinct-letter-in*[*OF* ‹$\neg\ rev\ w \in [a]*$›] **by** *blast*
have *eq*: $rev\ ([b] \cdot [a]^{@}m) = [a]^{@}m \cdot [b]$
by (*simp add*: *rev-pow*)
have $[b] \cdot [a]^{@}m \leq s\ w$
using ‹$[a]^{@}m \cdot [b] \cdot q = rev\ w$› **unfolding** *suf-rev-pref-iff eq lassoc*
using *prefI* **by** *blast*
thus *?thesis*
using ‹$b \neq a$› **by** *blast*
qed

lemma *sing-pow-exp*: **assumes** $w \in [a]*$ **shows** $w = [a]^{@}|w|$
proof–
obtain k **where** $[a]\ ^{@}\ k = w$
using *rootE*[*OF assms*]**.**
from *this*[*folded sing-pow-len*[*of a k*, *folded this*, *unfolded this*], *symmetric*]
show *?thesis***.**
qed

lemma *sing-power′*: **assumes** $w \in [a]*$ **and** $i < |w|$ **shows** $w\ !\ i = a$
using *sing-pow*[*OF* ‹$i < |w|$›, *of a*, *folded sing-pow-exp*[*OF* ‹$w \in [a]*$›]]**.**

lemma *rev-sing-power*: $x \in [a]* \Longrightarrow rev\ x = x$
unfolding *root-def* **using** *rev-pow rev-singleton-conv* **by** *metis*

lemma *lcp-letter-power*:
assumes $w \neq \varepsilon$ **and** $w \in [a]*$ **and** $[a]^{@}m \cdot [b] \leq p\ z$ **and** $a \neq b$
shows $w \cdot z \wedge_p z \cdot w = [a]^{@}m$
proof–
obtain $k\ z'$ **where** $w = [a]^{@}k\ z = [a]^{@}m \cdot [b] \cdot z'\ k \neq 0$
using ‹$w \in [a]*$› ‹$[a]^{@}m \cdot [b] \leq p\ z$› ‹$w \neq \varepsilon$› *nemp-pow*[*of* $[a]$]
unfolding *root-def*
by *auto*

hence *eq1*: $w \cdot z = [a]^{@}m \cdot ([a]^{@}k \cdot [b] \cdot z')$ **and** *eq2*: $z \cdot w = [a]^{@}m \cdot ([b] \cdot z' \cdot [a]^{@}k)$
by (*simp add*: ‹$w = [a]^{@}k$› ‹$z = [a]^{@}m \cdot [b] \cdot z'$› *pow-comm*)+
have *hd* $([a]^{@}k \cdot [b] \cdot z') = a$
using *hd-append2*[*OF* ‹$w \neq \varepsilon$›, *of* $[b] \cdot z'$,
unfolded ‹$w = (a \# \varepsilon)^{@}k$› *hd-sing-power*[*OF* ‹$k \neq 0$›, *of* a]].
moreover have $hd([b] \cdot z' \cdot [a]^{@}k) = b$
by *simp*
ultimately have $[a]^{@}k \cdot [b] \cdot z' \wedge_p [b] \cdot z' \cdot [a]^{@}k = \varepsilon$
by (*simp add*: ‹$a \neq b$› *lcp-distinct-hd*)
thus *?thesis* **using** *eq1 eq2 lcp-ext-left*[*of* $[a]^{@}m$ $[a]^{@}k \cdot [b] \cdot z'$ $[b] \cdot z' \cdot [a]^{@}k$]
by *simp*
qed

lemma *per-one*: **assumes** $w \leq p\ [a]^{\omega}$ **shows** $w \in [a]*$
proof–
have $w \leq p\ (a \# \varepsilon)\ ^{@}\ n \Longrightarrow n \neq 0 \Longrightarrow w \in [a]*$ **for** n
using *pref-sing-power*[*of w a*] *rootI*[*of* $[a]$ $|w|$] **by** *auto*
with *per-pref-ex*[*OF* ‹$w \leq p\ [a]^{\omega}$›]
show *?thesis* **by** *auto*
qed

lemma *per-one'*: $w \in [a]* \Longrightarrow w \leq p\ [a]^{\omega}$
by (*metis append-Nil2 not-Cons-self2 per-pref prefI root-def*)

lemma *per-sing-one*: **assumes** $w \neq \varepsilon$ $w \leq p\ [a]^{\omega}$ **shows** *periodN w 1*
using *root-periodN*[*OF* ‹$w \neq \varepsilon$› ‹$w \leq p\ [a]^{\omega}$›] **unfolding** *sing-len*[*of a*].

2.18 Border

A non-empty word $x \neq w$ is a *border* of a word w if it is both its prefix and suffix. This elementary property captures how much the word w overlaps with itself, and it is in the obvious way related to a period of w. However, in many cases it is much easier to reason about borders than about periods.

definition *border* :: *'a list* ⇒ *'a list* ⇒ *bool* (- ≤*b* - [*51,51*] *60*)
where [*simp*]: *border x w* = ($x \leq p\ w \wedge x \leq s\ w \wedge x \neq w \wedge x \neq \varepsilon$)

definition *bordered* :: *'a list* ⇒ *bool*
where [*simp*]: *bordered w* = (∃ *b*. *b* ≤*b w*)

lemma *borderI*[*intro*]: $x \leq p\ w \Longrightarrow x \leq s\ w \Longrightarrow x \neq w \Longrightarrow x \neq \varepsilon \Longrightarrow x \leq b\ w$
unfolding *border-def* **by** *blast*

lemma *borderD-pref*: $x \leq b\ w \Longrightarrow x \leq p\ w$
unfolding *border-def* **by** *blast*

lemma *borderD-spref*: $x \leq b\ w \Longrightarrow x <p\ w$
unfolding *border-def* **by** *simp*

lemma *borderD-suf*: $x \leq_b w \Longrightarrow x \leq_s w$
unfolding *border-def* **by** *blast*

lemma *borderD-ssuf*: $x \leq_b w \Longrightarrow x <_s w$
unfolding *border-def* **by** *blast*

lemma *borderD-nemp*: $x \leq_b w \Longrightarrow x \neq \varepsilon$
using *border-def* **by** *blast*

lemma *borderD-neq*: $x \leq_b w \Longrightarrow x \neq w$
unfolding *border-def* **by** *blast*

lemma *border-lq-nemp*: **assumes** $x \leq_b w$ **shows** $x^{-1>}w \neq \varepsilon$
using *assms borderD-spref lq-spref* **by** *blast*

lemma *border-rq-nemp*: **assumes** $x \leq_b w$ **shows** $w^{<-1}x \neq \varepsilon$
using *assms borderD-ssuf rq-ssuf* **by** *blast*

lemma *border-trans*[*trans*]: **assumes** $t \leq_b x$ $x \leq_b w$
shows $t \leq_b w$
using *assms* **unfolding** *border-def*
using *suffix-order.antisym pref-trans*[*of t x w*] *suf-trans*[*of t x w*] **by** *blast*

lemma *border-rev-conv*[*reversal-rule*]: *rev* $x \leq_b$ *rev* $w \longleftrightarrow x \leq_b w$
unfolding *border-def*
using *rev-is-Nil-conv*[*of x*] *rev-swap*[*of w*] *rev-swap*[*of x*]
suf-rev-pref-iff[*of x w*] *pref-rev-suf-iff*[*of x w*]
by *fastforce*

lemma *border-lq-comp*: $x \leq_b w \Longrightarrow (w^{<-1}x) \bowtie x$
unfolding *border-def* **using** *rq-suf ruler* **by** *blast*

lemmas *border-lq-suf-comp* = *border-lq-comp*[*reversed*]

2.18.1 The shortest border

lemma *border-len*: **assumes** $x \leq_b w$
shows $1 < |w|$ **and** $0 < |x|$ **and** $|x| < |w|$
proof–
show $|x| < |w|$
using *assms prefix-length-less* **unfolding** *border-def strict-prefix-def*
by *blast*
show $0 < |x|$
using *assms* **unfolding** *border-def* **by** *blast*
thus $1 < |w|$
using *assms* ‹$|x| < |w|$› **unfolding** *border-def*
by *linarith*
qed

lemma *borders-compare*: **assumes** $x \leq_b w$ **and** $x' \leq_b w$ **and** $|x'| < |x|$
shows $x' \leq_b x$
using *ruler-le*[*OF borderD-pref*[*OF* ‹$x' \leq_b w$›] *borderD-pref*[*OF* ‹$x \leq_b w$›] *less-imp-le-nat*[*OF* ‹$|x'| < |x|$›]]
suf-ruler-le[*OF borderD-suf*[*OF* ‹$x' \leq_b w$›] *borderD-suf*[*OF* ‹$x \leq_b w$›] *less-imp-le-nat*[*OF* ‹$|x'| < |x|$›]]
borderD-nemp[*OF* ‹$x' \leq_b w$›] ‹$|x'| < |x|$›
borderI **by** *blast*

lemma *unbordered-border*:
bordered $w \Longrightarrow \exists\ x.\ x \leq_b w \wedge \neg$ *bordered* x
proof (*induction* $|w|$ *arbitrary*: w *rule*: *less-induct*)
case *less*
obtain x' **where** $x' \leq_b w$
using *bordered-def less.prems* **by** *blast*
show *?case*
proof (*cases bordered* x')
assume $\neg$ *bordered* x'
thus *?case*
using ‹$x' \leq_b w$› **by** *blast*
next
assume *bordered* x'
from *less.hyps*[*OF border-len*(*3*)[*OF* ‹$x' \leq_b w$›] *this*]
show *?case*
using *border-trans*[*of - x' w*] ‹$x' \leq_b w$› **by** *blast*
qed
qed

lemma *unbordered-border-shortest*: $x \leq_b w \Longrightarrow \neg$ *bordered* $x \Longrightarrow\ y \leq_b w \Longrightarrow |x| \leq |y|$
using *bordered-def*[*of x*] *borders-compare*[*of x w y*] *not-le-imp-less*[*of* $|x|$ $|y|$] **by** *blast*

lemma *long-border-bordered*: **assumes** *long*: $|w| < |x| + |x|$ **and** *border*: $x \leq_b w$
shows $(w^{<-1}x)^{-1>}x \leq_b x$
proof–
define p s **where** $p = w^{<-1}x$ **and** $s = x^{-1>}w$
hence *eq*: $p{\cdot}x = x{\cdot}s$
using *assms* **unfolding** *border-def* **using** *lq-pref*[*of x w*] *rq-suf*[*of x w*] **by** *simp*
have $|p| < |x|$
using *p-def long*[*folded rq-len*[*OF borderD-suf*[*OF border*]]] **by** *simp*
have *px*: $p \cdot p^{-1>}x = x$ **and** *sx*: $p^{-1>}x \cdot s = x$
using *eqd-pref*[*OF eq less-imp-le*, *OF* ‹$|p| < |x|$›] **by** *blast+*
have $p^{-1>}x \neq \varepsilon$
using ‹$|p| < |x|$› *px* **by** *fastforce*
have $p \neq \varepsilon$
using *border-rq-nemp*[*OF border*] *p-def*
by *presburger*

have $p^{-1>}x \neq x$
using ‹$p \neq \varepsilon$› px **by** *force*
have $(p^{-1>}x) \leq_b x$
unfolding *border-def*
using *eqd-pref*[*OF eq less-imp-le*, *OF* ‹$|p| < |x|$›] ‹$p^{-1>}x \neq \varepsilon$› ‹$p^{-1>}x \neq x$› **by** *blast*
thus *?thesis*
unfolding *p-def*.
qed

thm *long-border-bordered*[*reversed*]

lemma *border-short-dec*: **assumes** *border*: $x \leq_b w$ **and** *short*: $|x| + |x| \leq |w|$
shows $x \cdot x^{-1>}(w^{<-1}x) \cdot x = w$
proof–
have *eq*: $x{\cdot}x^{-1>}w = w^{<-1}x{\cdot}x$
using *lq-pref*[*OF borderD-pref*[*OF border*]] *rq-suf*[*OF borderD-suf*[*OF border*]] **by** *simp*
have $|x| \leq |w^{<-1}x|$
using *short*[*folded rq-len*[*OF borderD-suf*[*OF border*]]] **by** *simp*
from *lq-pref*[*of x w*, *OF borderD-pref*[*OF border*], *folded conjunct2*[*OF eqd-pref*[*OF eq this*]]]
show *?thesis*.
qed

lemma *bordered-dec*: **assumes** *bordered w*
obtains *u v* **where** $u{\cdot}v{\cdot}u = w$ **and** $u \neq \varepsilon$
proof–
obtain *u* **where** $u \leq_b w$ **and** ¬ *bordered u*
using *unbordered-border*[*OF assms*] **by** *blast*
have $|u| + |u| \leq |w|$
using *long-border-bordered*[*OF* - ‹$u \leq_b w$›] ‹¬ *bordered u*› *bordered-def leI* **by** *blast*
from *border-short-dec*[*OF* ‹$u \leq_b w$› *this*, *THEN that*, *OF borderD-nemp*[*OF* ‹$u \leq_b w$›]]
show *thesis*.
qed

2.18.2 Relation to period and conjugation

lemma *border-conjug-eq*: $x \leq_b w \Longrightarrow (w^{<-1}x) \cdot w = w \cdot (x^{-1>}w)$
using *lq-rq-reassoc-suf*[*OF borderD-pref borderD-suf*, *symmetric*] **by** *blast*

lemma *border-per-root*: $x \leq_b w \Longrightarrow w \leq_p (w^{<-1}x) \cdot w$
using *border-conjug-eq* **by** *blast*

lemma *per-root-border*: **assumes** $|r| < |w|$ **and** $r \neq \varepsilon$ **and** $w \leq_p r \cdot w$
shows $r^{-1>}w \leq_b w$
proof

have $|r| \leq |w|$ **and** $r \leq_p w$
using *less-imp-le*[*OF* ‹$|r| < |w|$›] *pref-prod-long*[*OF* ‹$w \leq_p r \cdot w$›] **by** *blast+*
show $r^{-1>}w \leq_p w$
using *pref-lq*[*OF* ‹$r \leq_p w$› ‹$w \leq_p r \cdot w$›] **unfolding** *lq-triv*.
show $r^{-1>}w \leq_s w$
using ‹$r \leq_p w$› **by** *auto*
show $r^{-1>}w \neq w$
using ‹$r \leq_p w$› ‹$r \neq \varepsilon$› **by** *force*
show $r^{-1>}w \neq \varepsilon$
using *lq-pref*[*OF* ‹$r \leq_p w$›] ‹$|r| < |w|$› **by** *force*
qed

lemma *border-per*: **assumes** $x \leq_b w$ **shows** *periodN* w $(|w|-|x|)$
proof−
have $w = (w^{<-1}x)\cdot x$
using *rq-suf*[*OF borderD-suf*[*OF assms*]] **by** *simp*
have $w = x\cdot(x^{-1>}w)$
using *lq-pref*[*OF borderD-pref*[*OF assms*]] **by** *simp*
have *take*: *take* $(|w|-|x|)$ $w = w^{<-1}x$
using *borderD-suf*[*OF assms*] **by** *auto*
have *nemp*: *take* $(|w|-|x|)$ $w \neq \varepsilon$
using *assms* **by** *auto*
have $w \leq_p$ *take* $(|w|-|x|)$ $w \cdot w$
unfolding *take lassoc*[*of* $w^{<-1}x$ x $x^{-1>}w$, *folded* ‹$w = x \cdot x^{-1>}w$› ‹$w = w^{<-1}x \cdot x$›]
using *triv-pref*[*of* w $x^{-1>}w$].
thus *periodN* w $(|w|-|x|)$
unfolding *periodN-def period-root-def* **using** *nemp* **by** *blast*
qed

lemma *per-border*: **assumes** $n < |w|$ **and** *periodN* w n
shows *take* $(|w| - n)$ w $\leq_b w$
proof−
have *eq*: *take* $(|w| - n)$ w = *drop* n w
using *pref-take*[*OF* ‹*periodN* w n›[*unfolded per-shift*[*OF periodN-D1*[*OF* ‹*periodN* w n›] *per-positive*[*OF* ‹*periodN* w n›]]], *unfolded length-drop*].
have *take* $(|w| - n)$ $w \neq \varepsilon$
using ‹$n < |w|$› *take-eq-Nil* **by** *fastforce*
moreover have *take* $(|w| - n)$ $w \neq w$
using *periodN-D2*[*OF* ‹*periodN* w n›] ‹$n < |w|$› **unfolding** *take-all-iff*[*of* $|w|-n$ w] **by** *fastforce*
ultimately show *?thesis*
unfolding *border-def* **using** *take-is-prefix*[*of* $|w|-n$ w] *suffix-drop*[*of* n w, *folded eq*] **by** *blast*
qed

2.19 Primitive words

If a word w is not a non-trivial power of some other word, we say it is primitive.

definition *primitive* :: *'a list* ⇒ *bool*
  where *primitive u* = (∀ *r k*. $r^{@}k = u \longrightarrow k = 1$)

lemma *primI*: (⋀ *r k*. $r^{@}k = u \Longrightarrow k = 1$) ⟹ *primitive u*
  by (*simp add*: *primitive-def*)

lemma *prim-nemp*: *primitive u* ⟹ $u \neq \varepsilon$
proof−
  have $u = \varepsilon \Longrightarrow \varepsilon^{@}0 = u$ **by** *simp*
  thus *primitive u* ⟹ $u \neq \varepsilon$
    using *primitive-def zero-neq-one* **by** *blast*
qed

lemma *prim-exp-one*: *primitive u* ⟹ $r^{@}k = u \Longrightarrow k = 1$
  using *primitive-def* **by** *blast*

lemma *prim-exp-eq*: *primitive u* ⟹ $r^{@}k = u \Longrightarrow u = r$
  using *prim-exp-one power-one-right* **by** *blast*

lemma *prim-triv-root*: *primitive u* ⟹ $u \in t* \Longrightarrow t = u$
  using *prim-exp-eq* **unfolding** *root-def*
  unfolding *primitive-def root-def* **by** *fastforce*

lemma *prim-comm-exp*[*elim*]: **assumes** *primitive v* **and** $v{\cdot}u = u{\cdot}v$ **obtains** *k* **where** $u = v^{@}k$
  using ‹$v{\cdot}u = u{\cdot}v$›[*unfolded comm*] *prim-exp-eq*[*OF* ‹*primitive v*›] **by** *metis*

lemma *pow-non-prim*: *1* < *k* ⟹ ¬ *primitive* ($w^{@}k$)
  using *prim-exp-one* **by** *auto*

lemma *comm-non-prim*: **assumes** $u \neq \varepsilon$ $v \neq \varepsilon$ $u{\cdot}v = v{\cdot}u$ **shows** ¬ *primitive* ($u{\cdot}v$)
proof−
  obtain *t k m* **where** $u = t^{@}k$ $v = t^{@}m$
    using ‹$u{\cdot}v = v{\cdot}u$›[*unfolded comm*] **by** *blast*
  show *?thesis* **using** *pow-non-prim*[*of k+m t*]
    unfolding ‹$u = t^{@}k$› ‹$v = t^{@}m$› *pow-add-list*[*of t k m*]
   using *nemp-pow*[*OF* ‹$u \neq \varepsilon$›[*unfolded* ‹$u = t^{@}k$›]] *nemp-pow*[*OF* ‹$v \neq \varepsilon$›[*unfolded* ‹$v = t^{@}m$›]]
    by *linarith*
qed

lemma *prim-rotate-conv*: *primitive w* ⟷ *primitive* (*rotate n w*)
proof
  assume *primitive w* **show** *primitive* (*rotate n w*)
  proof (*rule primI*)

```
  fix r k assume r^@k = rotate n w
  obtain l where (rotate l r)^@k = w
    using rotate-back[of n w, folded ‹r^@k = rotate n w›, unfolded rotate-pow-comm]
by blast
  from prim-exp-one[OF ‹primitive w› this]
  show k = 1.
qed
next
  assume primitive (rotate n w)  show primitive w
   proof (rule primI)
     fix r k assume r^@k = w
      from prim-exp-one[OF ‹primitive (rotate n w)›, OF rotate-pow-comm[of n r
k, unfolded this, symmetric]]
     show k = 1.
   qed
 qed

lemma non-prim: assumes ¬ primitive w and w ≠ ε
  obtains r k where r ≠ ε and 1 < k and r^@k = w and w ≠ r
proof−
  from ‹¬ primitive w›[unfolded primitive-def]
  obtain r k where k ≠ 1 and r^@k = w  by blast
  have r ≠ ε
    using ‹w ≠ ε› ‹r^@k = w› emp-pow by blast
  have k ≠ 0
    using ‹w ≠ ε› ‹r^@k = w› pow-zero[of r] by meson
  have w ≠ r
    using ‹k ≠ 1›[folded eq-pow-exp[OF ‹r ≠ ε›, of k 1, unfolded ‹r ^@ k = w›]] by
simp
  show thesis
    using that[OF ‹r ≠ ε› - ‹r^@k = w› ‹w ≠ r›] ‹k ≠ 0› ‹k ≠ 1› less-linear by
blast
qed

lemma prim-no-rotate: assumes primitive w and 0 < n and n < |w|
  shows rotate n w ≠ w
proof
  assume rotate n w = w
  have take n w · drop n w = drop n w · take n w
    using rotate-append[of take n w drop n w]
     unfolding take-len[OF less-imp-le-nat[OF ‹n < |w|›]] append-take-drop-id
‹rotate n w = w›.
  have take n w ≠ ε drop n w ≠ ε
    using ‹0 < n› ‹n < |w|› by auto+
  from ‹primitive w› show False
    using comm-non-prim[OF ‹take n w ≠ ε› ‹drop n w ≠ ε› ‹take n w · drop n w
= drop n w · take n w›, unfolded append-take-drop-id]
    by simp
qed
```

lemma *no-rotate-prim*: **assumes** $w \neq \varepsilon$ **and** $\bigwedge n.\ 0 < n \Longrightarrow n < |w| \Longrightarrow$ *rotate* $n\ w \neq w$
shows *primitive* w
proof (*rule ccontr*)
assume $\neg$ *primitive* w
from *non-prim*[*OF this* ‹$w \neq \varepsilon$›]
obtain $r\ l$ **where** $r \neq \varepsilon$ **and** $1 < l$ **and** $r^{@}l = w$ **and** $w \neq r$ **by** *blast*
have *rotate* $|r|\ w = w$
using *rotate-root-self*[*of r l*, *unfolded* ‹$r^{@}l = w$›].
moreover have $0 < |r|$
by (*simp add*: ‹$r \neq \varepsilon$›)
moreover have $|r| < |w|$
unfolding *pow-len*[*of r l*, *unfolded* ‹$r^{@}l = w$›] **using** ‹$1 < l$› ‹$0 < |r|$› **by** *auto*
ultimately show *False*
using *assms*(*2*) **by** *blast*
qed

corollary *prim-iff-rotate*: **assumes** $w \neq \varepsilon$ **shows**
primitive $w \longleftrightarrow (\forall\ n.\ 0 < n \wedge n < |w| \longrightarrow$ *rotate* $n\ w \neq w)$
using *no-rotate-prim*[*OF* ‹$w \neq \varepsilon$›] *prim-no-rotate* **by** *blast*

lemma *prim-sing*: *primitive* $[a]$
using *prim-iff-rotate*[*of* $[a]$] **by** *fastforce*

lemma *prim-rev-iff*[*reversal-rule*]: *primitive* (*rev u*) $\longleftrightarrow$ *primitive u*
unfolding *primitive-def*[*reversed*] **using** *primitive-def*..

2.20 Primitive root

Given a non-empty word w which is not primitive, it is natural to look for the shortest u such that $w = u^k$. Such a word is primitive, and it is the primitive root of w.

definition *primitive-rootP* :: *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *bool* (- $\in_p$ - $*$ [*51*,*51*] *60*)
where *primitive-rootP x r* $= (x \neq \varepsilon \wedge x \in r* \wedge$ *primitive r*)

lemma *prim-rootD* [*dest*]: $x \in_p r* \Longrightarrow x \in r*$
unfolding *primitive-rootP-def* **by** (*elim conjE*)

lemma *prim-rootI* [*intro*]: $u \neq \varepsilon \Longrightarrow\ u \in r* \Longrightarrow$ *primitive r* $\Longrightarrow u \in_p r*$
unfolding *primitive-rootP-def* **by** (*intro conjI*)

lemma *prim-root-rev-conv* [*reversal-rule*]: *rev x* $\in_p$ *rev r*$* \longleftrightarrow\ x \in_p r*$
unfolding *primitive-rootP-def*[*reversed*] **using** *primitive-rootP-def*..

fun *primitive-root* :: *'a list* $\Rightarrow$ *'a list* (ϱ) **where** *primitive-root x* $=$ (*THE r*. $x \in_p r*$)

lemma *primrootE*: **assumes** $x \in_p r*$
obtains k **where** $k \neq 0$ **and** $r^{@}k = x$
using *assms* **unfolding** *primitive-rootP-def root-def* **using** *nemp-pow*[*of r*] **by** *auto*

lemma *primroot-of-root*: $[\![x \neq \varepsilon; x \in u*; u \in_p r*]\!] \Longrightarrow x \in_p r*$
unfolding *primitive-rootP-def* **using** *root-trans* **by** *blast*

lemma *comm-prim*: **assumes** *primitive r* **and** *primitive s* **and** $r{\cdot}s = s{\cdot}r$
shows $r = s$
using ‹$r{\cdot}s = s{\cdot}r$›[*unfolded comm*] *assms*[*unfolded primitive-def*, *rule-format*] **by** *metis*

lemma *primroot-ex*: **assumes** $x \neq \varepsilon$ **shows** $\exists\ r\ k.\ \ x \in_p r* \wedge k \neq 0 \wedge x = r^{@}k$
using ‹$x \neq \varepsilon$›
proof(*induction* $|x|$ *arbitrary*: *x rule*: *less-induct*)
case *less*
then show $\exists\ r\ k.\ \ x \in_p r* \wedge k \neq 0 \wedge x = r^{@}k$
proof (*cases primitive x*)
assume ¬ *primitive x*
from *non-prim*[*OF this* ‹$x \neq \varepsilon$›]
obtain $r\ l$ **where** $r \neq \varepsilon$ **and** $1 < l$ **and** $r^{@}l = x$ **and** $x \neq r$ **by** *blast*
then obtain $pr\ k$ **where** $r \in_p pr*\ k \neq 0\ r = pr^{@}k$
using ‹$x \neq \varepsilon$› *less.hyps rootI root-shorter* **by** *blast*
hence $x \in_p pr*$
using ‹$r^{@}\ l = x$› *less.prems primroot-of-root rootI* **by** *blast*
show $\exists\ r\ k.\ \ x \in_p r* \wedge k \neq 0 \wedge x = r^{@}k$
using ‹$x \in_p pr*$›[*unfolded primitive-rootP-def root-def*]
‹$x \in_p pr\ *$› *nemp-pow* **by** *blast*
next
assume *primitive x*
have $x \in_p x*$
by (*simp add*: ‹*primitive x*› *less.prems prim-rootI self-root*)
thus $\exists\ r\ k.\ \ x \in_p r* \wedge k \neq 0 \wedge x = r^{@}k$
by *force*
qed
qed

lemma *primroot-exE*: **assumes**$x \neq \varepsilon$
obtains $r\ k$ **where** *primitive r* **and** $k \neq 0$ **and** $x = r^{@}k$
using *assms primitive-rootP-def primroot-ex*[*OF* ‹ $x \neq \varepsilon$›] **by** *blast*

Uniqueness of the primitive root follows from the following lemma

lemma *primroot-unique*: **assumes** $u \in_p r*$ **shows** $\varrho\ u = r$
proof−
obtain kr **where** $kr \neq 0$ **and** $r^{@}kr = u$
using *primrootE*[*OF* ‹$u \in_p r*$›].
have $u \in_p s* \Longrightarrow s = r$ **for** s

proof–
fix s **assume** $u \in_p s*$
obtain ks **where** $ks \neq 0$ **and** $s^@ks = u$
using *primrootE*[*OF* ‹$u \in_p s*$›].
obtain t **where** $s \in t*$ **and** $r \in t*$
using *comm-rootE*[*OF pow-comm-comm*[*of r kr s ks*, *OF* - ‹$kr \neq 0$›, *unfolded* ‹$r^@kr = u$› ‹$s^@ks = u$›, *OF refl*]].
have *primitive r* **and** *primitive s*
using ‹$u \in_p r *$› ‹$u \in_p s *$› *primitive-rootP-def* **by** *blast+*
from *prim-exp-eq*[*OF* ‹*primitive r*›, *of t*] *prim-exp-eq*[*OF* ‹*primitive s*›, *of t*]
show $s = r$
using *rootE*[*OF* ‹$s \in t*$›, *of s=r*] *rootE*[*OF* ‹$r \in t*$›, *of* $r = t$] **by** *fastforce*
qed
from *the-equality*[*of* λ r. $u \in_p r*$,*OF* ‹$u \in_p r*$› *this*]
show $\varrho\ u = r$
by *auto*
qed

lemma *prim-self-root*: *primitive* $x \Longrightarrow \varrho\ x\ = x$
using *prim-nemp prim-rootI primroot-unique self-root* **by** *blast*

Existence and uniqueness of the primitive root justifies the function ϱ: it indeed yields the primitive root of a nonempty word.

lemma *primroot-is-primroot*[*intro*]: **assumes** $x \neq \varepsilon$ **shows** $x \in_p (\varrho\ x)*$
using *primroot-ex*[*OF* ‹$x \neq \varepsilon$›] *primroot-unique*[*of x*]
by *force*

lemma *primroot-is-root*[*intro*]: $x \neq \varepsilon \Longrightarrow x \in (\varrho\ x)*$
using *primroot-is-primroot* **by** *auto*

lemma *primrootI* [*intro*]: **assumes** $x \neq \varepsilon$ **shows** *primroot-prim*: *primitive* $(\varrho\ x)$ **and** *primroot-nemp*: $\varrho\ x \neq \varepsilon$
using *assms prim-nemp primitive-rootP-def* **by** *blast+*

lemma *primroot-root*: **assumes** $u \neq \varepsilon$ $u \in q*$ **shows** $\varrho\ q = \varrho\ u$
using *primroot-unique*[*OF primroot-of-root*[*OF* ‹$u \neq \varepsilon$› ‹$u \in q*$› *primroot-is-primroot*, *OF root-nemp*[*OF* ‹$u \neq \varepsilon$› ‹$u \in q*$›]], *symmetric*].

lemma *primroot-len-mult*: **assumes** $u \neq \varepsilon$ $u \in q*$ **obtains** k **where** $|q| = k*|\varrho\ u|$
using *primroot-is-primroot*[*OF root-nemp*[*OF* ‹$u \neq \varepsilon$› ‹$u \in q*$›], *unfolded primroot-root*[*OF* ‹$u \neq \varepsilon$› ‹$u \in q*$›]
primitive-rootP-def] *root-len*[*of q* ϱ *u*] **by** *blast*

lemma *primroot-shorter-root*: **assumes** $u \neq \varepsilon$ $u \in q*$ **shows** $|\varrho\ u| \leq |q|$
using *quotient-smaller*[*OF root-nemp*[*OF* ‹$u \neq \varepsilon$› ‹$u \in q*$›, *folded length-0-conv*], *of* - $|\varrho\ u|$]
primroot-len-mult[*OF* ‹$u \neq \varepsilon$› ‹$u \in q*$›] **by** *blast*

lemma *primroot-shortest-root*: **assumes** $u \neq \varepsilon$ **shows** $|\varrho\ u| = (LEAST\ d.\ (\exists\ r.$

$(u \in r*) \wedge |r| = d))$
using *Least-equality*[*of* $\lambda\ k.\ (\exists\ r.\ (u \in r*) \wedge |r| = k)\ |\varrho\ u|$]
proof
show $\exists r.\ u \in r* \wedge |r| = |\varrho\ u|$
using *assms primitive-rootP-def primroot-is-primroot* **by** *blast*
show $\bigwedge y.\ \exists r.\ u \in r* \wedge |r| = y \Longrightarrow |\varrho\ u| \leq y$
using *assms primroot-shorter-root* **by** *fastforce*
qed

lemma *primroot-shorter-eq*: $u \neq \varepsilon \Longrightarrow |\varrho\ u| \leq |u|$
using *primroot-shorter-root self-root* **by** *auto*

lemma *primroot-take*: **assumes** $u \neq \varepsilon$ **shows** $\varrho\ u = (take\ (\ |\varrho\ u|\ )\ u)$
proof–
obtain k **where** $(\varrho\ u)^{@}k = u$ **and** $k \neq 0$
using *primrootE*[*OF primroot-is-primroot*[*OF* ‹$u \neq \varepsilon$›]]**.**
show $\varrho\ u = (take\ (\ |\varrho\ u|\ )\ u)$
using *take-root*[*of* - $(\varrho\ u)$, *OF* ‹$k \neq 0$›, *unfolded* ‹$(\varrho\ u)^{@}k = u$›]**.**
qed

lemma *primroot-take-shortest*: **assumes** $u \neq \varepsilon$ **shows** $\varrho\ u = (take\ (LEAST\ d.\ (\exists\ r.\ (u \in r*) \wedge |r| = d))\ u)$
using *primroot-take*[*OF assms*, *unfolded primroot-shortest-root*[*OF assms*]]**.**

lemma *primroot-rotate-comm*: **assumes** $w \neq \varepsilon$ **shows** $\varrho\ (rotate\ n\ w) = rotate\ n\ (\varrho\ w)$
proof–
obtain l **where** $w = (\varrho\ w)^{@}l$
using *pow-zero primrootE primroot-is-primroot* **by** *metis*
hence $rotate\ n\ w \in (rotate\ n\ (\varrho\ w))*$
using *rotate-pow-comm root-def* **by** *metis*
moreover have $rotate\ n\ w \neq \varepsilon$
using *assms* **by** *auto*
moreover have $primitive\ (rotate\ n\ (\varrho\ w))$
using *assms prim-rotate-conv primitive-rootP-def primroot-is-primroot* **by** *blast*
ultimately have $rotate\ n\ w \in_p (rotate\ n\ (\varrho\ w))*$
unfolding *primitive-rootP-def* **by** *blast*
thus *?thesis*
using *primroot-unique* **by** *blast*
qed

lemma *primrootI1* [*intro*]: **assumes** *pow*: $u = r^{@}(Suc\ k)$ **and** *prim*: *primitive* r **shows** $\varrho\ u = r$
proof–
have $u \neq \varepsilon$
using *pow prim prim-nemp* **by** *auto*
have $u \in r*$
using *pow rootI* **by** *blast*

show $\varrho\ u = r$
using *primroot-unique*[*OF prim-rootI*[*OF* ‹$u \neq \varepsilon$› ‹$u \in r*$› ‹*primitive r*›]].
qed

lemma *prim-root-power* [*elim*]: **assumes** $x \neq \varepsilon$ **obtains** i **where** $(\varrho\ x)^{@}(Suc\ i) = x$
using *prim-rootD*[*OF primroot-is-primroot*[*OF* ‹$x \neq \varepsilon$›], *unfolded root-def*] *assms pow-zero*[*of* $\varrho\ x$] *not0-implies-Suc*
by *metis*

lemma *prim-root-cases*: **obtains** $u = \varepsilon \mid$ *primitive* $u \mid |\varrho\ u| < |u|$
using *primroot-is-primroot*[*THEN prim-rootD*[*of* $u\ \varrho\ u$]]
primroot-prim[*of* u] *root-shorter*[*of* $u\ \varrho\ u$] **by** *fastforce*

We also have the standard characterization of commutation for nonempty words.

theorem *comm-primroots*: **assumes** $u \neq \varepsilon\ v \neq \varepsilon$ **shows** $u \cdot v = v \cdot u \longleftrightarrow \varrho\ u = \varrho\ v$
proof
assume $u \cdot v = v \cdot u$
then obtain t **where** $u \in t*$ **and** $v \in t*$
using *comm-root* **by** *blast*
show $\varrho\ u = \varrho\ v$
using *primroot-root*[*OF* ‹$v \neq \varepsilon$› ‹$v \in t*$›, *unfolded primroot-root*[*OF* ‹$u \neq \varepsilon$› ‹$u \in t*$›]].
next
assume $\varrho\ u = \varrho\ v$
show $u \cdot v = v \cdot u$
using *primroot-is-primroot*[*OF* ‹$u \neq \varepsilon$›, *unfolded* ‹$\varrho\ u = \varrho\ v$›] *primroot-is-primroot*[*OF* ‹$v \neq \varepsilon$›] **unfolding** *primitive-rootP-def*
comm-root **by** *blast*
qed

lemma *prim-comm-short-emp*: **assumes** *primitive* p **and** $u{\cdot}p{=}p{\cdot}u$ **and** $|u| < |p|$
shows $u = \varepsilon$
proof (*rule ccontr*)
assume $u \neq \varepsilon$
from ‹$u \cdot p = p \cdot u$›
have $\varrho\ u = \varrho\ p$
unfolding *comm-primroots*[*OF* ‹$u \neq \varepsilon$› *prim-nemp*, *OF* ‹*primitive p*›].
have $\varrho\ u = p$
using *prim-self-root*[*OF* ‹*primitive p*›, *folded* ‹$\varrho\ u = \varrho\ p$›].
from ‹$|u| < |p|$›[*folded this*]
show *False*
using *primroot-shorter-eq*[*OF* ‹$u \neq \varepsilon$›] **by** *auto*
qed

2.21 Conjugation

Two words x and y are conjugated if one is a rotation of the other. Or, equivalently, there exists z such that

$$xz = zy.$$

definition *conjugate* (- $\sim$ - [*50*,*50*] *51*) **where** $u \sim v \equiv \exists\, r\ s.\ r \cdot s = u \wedge s \cdot r = v$

lemma *conjug-rev-conv* [*reversal-rule*]: *rev u* $\sim$ *rev v* $\longleftrightarrow$ $u \sim v$
unfolding *conjugate-def*[*reversed*] **using** *conjugate-def* **by** *blast*

lemma *conjug-rotate-iff*: $u \sim v \longleftrightarrow (\exists\ n.\ v = rotate\ n\ u)$
unfolding *conjugate-def*
using *rotate-drop-take*[*of* - *u*] *takedrop*[*of* - *u*] *rotate-append*
by *metis*

lemma *conjugI* [*intro*]: $r \cdot s = u \Longrightarrow s \cdot r = v \Longrightarrow u \sim v$
unfolding *conjugate-def* **by** (*intro exI conjI*)

lemma *conjugI'* [*intro*!]: $r \cdot s \sim s \cdot r$
unfolding *conjugate-def* **by** (*intro exI conjI*, *standard*+)

lemma *conjugE* [*elim*]:
assumes $u \sim v$
obtains *r s* **where** $r \cdot s = u$ **and** $s \cdot r = v$
using *assms* **unfolding** *conjugate-def* **by** (*elim exE conjE*)

lemma *conjugE1* [*elim*]:
assumes $u \sim v$
obtains *r* **where** $u \cdot r = r \cdot v$
proof −
obtain *r s* **where** *u*: $r \cdot s = u$ **and** *v*: $s \cdot r = v$ **using** *assms*..
have $u \cdot r = r \cdot v$ **unfolding** *u*[*symmetric*] *v*[*symmetric*] **using** *rassoc*.
then show *thesis* **by** *fact*
qed

lemma *conjug-refl*: $u \sim u$
by *standard*+

lemma *conjug-sym* [*sym*]: $u \sim v \Longrightarrow v \sim u$
by (*elim conjugE*, *intro conjugI*) *assumption*+

lemma *conjug-nemp-iff*: $u \sim v \Longrightarrow u = \varepsilon \longleftrightarrow v = \varepsilon$
by (*elim conjugE1*, *intro iffI*) *simp*+

lemma *conjug-len*: $u \sim v \implies |u| = |v|$
by (*elim conjugE*, *hypsubst*, *rule swap-len*)

lemma *pow-conjug*:
assumes *eq*: $t^{@}i \cdot r \cdot u = t^{@}k$ **and** t: $r \cdot s = t$
shows $u \cdot t^{@}i \cdot r = (s \cdot r)^{@}k$
proof −
have $t^{@}i \cdot r \cdot u \cdot t^{@}i \cdot r = t^{@}i \cdot t^{@}k \cdot r$ **unfolding** *eq*[*unfolded lassoc*] *lassoc append-same-eq pow-comm*..
also have ... $= t^{@}i \cdot r \cdot (s \cdot r)^{@}k$ **unfolding** *conjug-pow*[*OF rassoc*, *symmetric*] *t*..
finally show $u \cdot t^{@}i \cdot r = (s \cdot r)^{@}k$ **unfolding** *same-append-eq*.
qed

The solution of the equation

$$xz = zy$$

is given by the next lemma.

lemma *conjug-eqE* [*elim*, *consumes 2*]:
assumes *eq*: $x \cdot z = z \cdot y$ **and** $x \neq \varepsilon$
obtains $u\ v\ k$ **where** $u \cdot v = x$ **and** $v \cdot u = y$ **and** $(u \cdot v)^{@}k \cdot u = z$ **and** $v \neq \varepsilon$
proof −
have $z \leq p\ x \cdot z$ **using** *eq*[*symmetric*]..
from *this* ‹$x \neq \varepsilon$› **have** $z \leq p\ x^{\omega}$..
then obtain $k\ u$ **where** $x^{@}k \cdot u = z$ **and** $u <p\ x$..
from ‹$u <p\ x$› **obtain** v **where** x: $u \cdot v = x$ **and** $v \neq \varepsilon$..
have z: $(u{\cdot}v)^{@}k \cdot u = z$ **unfolding** x ‹$x^{@}k \cdot u = z$›..
have $z \cdot y = (u{\cdot}v) \cdot ((u{\cdot}v)^{@}k \cdot u)$ **unfolding** z **unfolding** x *eq*..
also have ... $= (u{\cdot}v)^{@}k \cdot u \cdot (v \cdot u)$ **unfolding** *lassoc pow-commutes-list*[*symmetric*]..
finally have y: $v \cdot u = y$ **unfolding** z[*symmetric*] *rassoc same-append-eq*..
from $x\ y\ z$ ‹$v \neq \varepsilon$› **show** *thesis*..
qed

theorem *conjugation*: **assumes** $x{\cdot}z = z{\cdot}y$ **and** $x \neq \varepsilon$
shows $\exists\ u\ v\ k.\ u \cdot v = x \wedge v \cdot u\ = y \wedge (u \cdot v)^{@}k \cdot u = z$
using *assms* **by** *blast*

lemma *conjug-eq-prim-rootE* [*elim*, *consumes 2*]:
assumes *eq*: $x \cdot z = z \cdot y$ **and** $x \neq \varepsilon$
obtains $r\ s\ i\ n$ **where**
$(r \cdot s)^{@}Suc\ i = x$ **and**
$(s \cdot r)^{@}Suc\ i = y$ **and**
$(r \cdot s)^{@}n \cdot r = z$ **and**
$s \neq \varepsilon$ **and** *primitive* $(r \cdot s)$
proof −
from ‹$x \neq \varepsilon$› **obtain** i **where** $(\varrho\ x)^{@}(Suc\ i) = x$..
also have $z \leq p\ x^{\omega}$ **using** *prefI*[*OF* ‹$x \cdot z = z \cdot y$›[*symmetric*]] ‹$x \neq \varepsilon$›..
finally have $z \leq p\ (\varrho\ x)^{\omega}$ **by** (*elim per-drop-exp*)
then obtain $n\ r$ **where** $(\varrho\ x)^{@}n \cdot r = z$ **and** $r <p\ \varrho\ x$..
from ‹$r <p\ \varrho\ x$› **obtain** s **where** $r \cdot s = \varrho\ x$ **and** $s \neq \varepsilon$..
define j **where** $j = Suc\ i$
have x: $(r{\cdot}s)^{@}j = x$ **unfolding** ‹$r \cdot s = \varrho\ x$› ‹$j = Suc\ i$› ‹$(\varrho\ x)^{@}(Suc\ i) = x$›..

have z: $(r{\cdot}s)^{@}n \cdot r = z$ **unfolding** $\langle r \cdot s = \varrho\ x\rangle$ $\langle(\varrho\ x)^{@}n \cdot r = z\rangle$**..**
have $z \cdot y = ((r{\cdot}s)^{@}j \cdot (r{\cdot}s)^{@}n) \cdot r$ **using** *eq*[*symmetric*] **unfolding** *rassoc x z*.
also have $(r{\cdot}s)^{@}j \cdot (r{\cdot}s)^{@}n = (r{\cdot}s)^{@}n \cdot (r{\cdot}s)^{@}j$ **using** *pow-comm*.
also have $\ldots \cdot r = (r{\cdot}s)^{@}n \cdot r \cdot (s{\cdot}r)^{@}j$ **unfolding** *rassoc* **unfolding** *shift-pow*..
finally have *y*: $y = (s{\cdot}r)^{@}j$ **unfolding** *z*[*symmetric*] *rassoc cancel*.
from $\langle x \neq \varepsilon\rangle$ **have** *primitive* $(r \cdot s)$ **unfolding** $\langle r \cdot s = \varrho\ x\rangle$..
with *that x y z* $\langle s \neq \varepsilon\rangle$ **show** *thesis* **unfolding** $\langle j = Suc\ i\rangle$ **by** *blast*
qed

lemma *conjug-eq-prim-root*:
assumes $x \cdot z = z \cdot y$ **and** $x \neq \varepsilon$
shows $\exists\ r\ s\ i\ n.\ (r \cdot s)^{@}(Suc\ i) = x \wedge (s \cdot r)^{@}(Suc\ i) = y \wedge (r \cdot s)^{@}n \cdot r = z \wedge s \neq \varepsilon \wedge primitive\ (r \cdot s)$
using *conjug-eq-prim-rootE*[*OF assms*, *of ?thesis*] **by** *blast*

lemma *conjugI1* [*intro*]:
assumes *eq*: $u \cdot r = r \cdot v$
shows $u \sim v$
proof (*cases*)
assume $u = \varepsilon$
have $v = \varepsilon$ **using** *eq* **unfolding** $\langle u = \varepsilon\rangle$ **by** *simp*
show $u \sim v$ **unfolding** $\langle u = \varepsilon\rangle$ $\langle v = \varepsilon\rangle$ **using** *conjug-refl*.
next
assume $u \neq \varepsilon$
show $u \sim v$ **using** *eq* $\langle u \neq \varepsilon\rangle$ **by** (*cases rule*: *conjug-eqE*, *intro conjugI*)
qed

lemma *conjug-trans* [*trans*]:
assumes *uv*: $u \sim v$ **and** *vw*: $v \sim w$
shows $u \sim w$
using *assms* **unfolding** *conjug-rotate-iff* **using** *rotate-rotate* **by** *blast*

lemma *conjug-trans'*: **assumes** *uv'*: $u \cdot r = r \cdot v$ **and** *vw'*: $v \cdot s = s \cdot w$ **shows** $u \cdot (r \cdot s) = (r \cdot s) \cdot w$
proof −
have $u \cdot (r \cdot s) = (r \cdot v) \cdot s$ **unfolding** *uv'*[*symmetric*] *rassoc*..
also have $\ldots = r \cdot (s \cdot w)$ **unfolding** *vw'*[*symmetric*] *rassoc*..
finally show $u \cdot (r \cdot s) = (r \cdot s) \cdot w$ **unfolding** *rassoc*.
qed

lemma *nconjug-neq*: $\neg\ u \sim v \Longrightarrow u \neq v$
by *blast*

lemma *prim-conjug*:
assumes *prim*: *primitive u* **and** *conjug*: $u \sim v$
shows *primitive v*
proof −
have $v \neq \varepsilon$ **using** *prim-nemp*[*OF prim*] **unfolding** *conjug-nemp-iff*[*OF conjug*].
from *conjug*[*symmetric*] **obtain** t **where** $v \cdot t = t \cdot u$..

from *this* ‹$v \neq \varepsilon$› **obtain** $r\ s\ i$ **where**
v: $(r \cdot s)^{@}(Suc\ i) = v$ **and** u: $(s \cdot r)^{@}(Suc\ i) = u$ **and** *prim'*: *primitive* $(r \cdot s)$**..**
have $r \cdot s = v$ **using** v **unfolding** *prim-exp-one*[*OF prim u*] *pow-one-id***.**
show *primitive* v **using** *prim'* **unfolding** ‹$r \cdot s = v$›**.**
qed

lemma *root-conjug*: $u \leq p\ r \cdot u \Longrightarrow u^{-1>}(r \cdot u) \sim r$
using *conjugI1 conjug-sym lq-pref* **by** *metis*

lemma *conjug-prim-root*:
assumes *conjug*: $u \sim v$ **and** $u \neq \varepsilon$
shows $\varrho\ u \sim \varrho\ v$
proof −
from *conjug* **obtain** t **where** $u \cdot t = t \cdot v$**..**
from *this* ‹$u \neq \varepsilon$› **obtain** $r\ s\ i$ **where**
u: $(r \cdot s)^{@}(Suc\ i) = u$ **and** v: $(s \cdot r)^{@}(Suc\ i) = v$ **and** *prim*: *primitive* $(r \cdot s)$**..**
have *rs*: $\varrho\ u = r \cdot s$ **and** *sr*: $\varrho\ v = s \cdot r$
using *prim prim-conjug u v* **by** *blast+*
show $\varrho\ u \sim \varrho\ v$ **using** *conjugI'* **unfolding** *rs sr***.**
qed

lemma *conjug-add-exp*: $u \sim v \Longrightarrow\ u^{@}k \sim v^{@}k$
by (*elim conjugE1*, *intro conjugI1*, *rule conjug-pow*)

lemma *conjug-prim-root-iff*:
assumes *nemp*:$u \neq \varepsilon$ **and** *len*: $|u| = |v|$
shows $\varrho\ u\ \sim \varrho\ v \longleftrightarrow u \sim v$
proof
show $u \sim v \Longrightarrow \varrho\ u\ \sim \varrho\ v$ **using** *conjug-prim-root*[*OF* - *nemp*]**.**
assume *conjug*: $\varrho\ u\ \sim \varrho\ v$
have $v \neq \varepsilon$ **using** *nemp-len*[*OF nemp*] **unfolding** *len length-0-conv***.**
with *nemp* **obtain** $k\ l$ **where** *roots*: $(\varrho\ u)^{@}k = u\ (\varrho\ v)^{@}l = v$ **by** (*elim prim-root-power*)
have $|(\varrho\ u)^{@}k| = |(\varrho\ v)^{@}l|$ **using** *len* **unfolding** *roots***.**
then have $k = l$ **using** *primroot-nemp*[*OF* ‹$v \neq \varepsilon$›]
unfolding *pow-len conjug-len*[*OF conjug*] **by** *simp*
show $u \sim v$ **using** *conjug-add-exp*[*OF conjug*, *of l*] **unfolding** *roots*[*unfolded* ‹$k = l$›]**.**
qed

lemma *fac-pow-pref-conjug*:
assumes $u \leq f\ t^{@}k$
obtains t' **where** $t \sim t'$ **and** $u \leq p\ t'^{@}k$
proof (*cases* $u = \varepsilon$)
assume $u \neq \varepsilon$
obtain $p\ q$ **where** *eq*: $p \cdot u \cdot q = t^{@}k$ **using** *facE'*[*OF assms*]**.**
obtain $i\ r$ **where** $i < k$ **and** $r <p\ t$ **and** p: $t^{@}i \cdot r = p$
using *pref-mod-power*[*OF sprefI1'*[*OF eq pref-nemp*[*OF* ‹$u \neq \varepsilon$›]]]**.**
from ‹$r <p\ t$› **obtain** s **where** t: $r \cdot s = t$**..**

have *eq′*: $t^{@}i \cdot r \cdot (u \cdot q) = t^{@}k$ **using** *eq* **unfolding** *lassoc p*.
have $u \leq p\ (s \cdot r)^{@}k$ **using** *pow-conjug*[*OF eq′ t*] **unfolding** *rassoc*..
with *conjugI′*[*of r s*] **show** *thesis* **unfolding** *t*..
qed *blast*

lemma *fac-pow-len-conjug*: **assumes** $|u| = |v|$ **and** $u \leq f\ v^{@}k$ **shows** $v \sim u$
proof−
obtain t **where** $v \sim t$ **and** $u \leq p\ t^{@}k$
using *fac-pow-pref-conjug assms* **by** *blast*
have $u = t$
using *pref-equal*[*OF pref-prod-root*[*OF* ‹$u \leq p\ t^{@}k$›] *conjug-len*[*OF* ‹$v \sim t$›,*folded* ‹$|u| = |v|$›]].
from ‹$v \sim t$›[*folded this*]
show $v \sim u$.
qed

lemma *border-conjug*: $x \leq b\ w \Longrightarrow w^{<-1}x \sim\ x^{-1>}w$
using *border-conjug-eq conjugI1* **by** *blast*

lemmas *fac-pow-suf-conjug* = *fac-pow-pref-conjug*[*reversed*]

end

theory *Submonoids*
imports *CoWBasic*
begin

Chapter 3

Submonoids of a free monoid

This chapter deals with properties of submonoids of a free monoid, that is, with monoids of words. See more in Chapter 1 of [4].

3.1 Hull

First, we define the hull of a set of words, that is, the monoid generated by them.

inductive-set *hull* :: *'a list set* ⇒ *'a list set* (⟨-⟩)
  for *G* **where**
   emp-in: $\varepsilon \in \langle G \rangle$ |
   prod-cl: $w1 \in G \Longrightarrow w2 \in \langle G \rangle \Longrightarrow w1 \cdot w2 \in \langle G \rangle$

lemmas [*intro*] = *hull.intros*

lemma *hull-closed*: $w1 \in \langle G \rangle \Longrightarrow w2 \in \langle G \rangle \Longrightarrow w1 \cdot w2 \in \langle G \rangle$
  by (*rule hull.induct*[*of w1 G* λ *x*. (*x*·*w2*)∈⟨*G*⟩]) *auto*+

lemma *gen-in* [*intro*]: $w \in G \Longrightarrow w \in \langle G \rangle$
  using *hull.prod-cl* **by** *fastforce*

lemma *hull-induct*: **assumes** $x \in \langle G \rangle$ *P* ε ⋀*w*. $w \in G \Longrightarrow P\ w$
⋀*w1 w2*. $w1 \in \langle G \rangle \Longrightarrow P\ w1 \Longrightarrow w2 \in \langle G \rangle \Longrightarrow P\ w2 \Longrightarrow P\ (w1 \cdot w2)$ **shows** *P x*
  using *hull.induct*[*of - - P*, *OF* ‹*x* ∈ ⟨*G*⟩› ‹*P* ε›]
      assms **by** (*simp add*: *gen-in*)

lemma *genset-sub*: $G \subseteq \langle G \rangle$
   using *gen-in* **..**

lemma *in-lists-conv-set-subset*: *set ws* ⊆ *G* ⟷ *ws* ∈ *lists G*
  by *blast*

lemma *concat-in-hull* [*intro*]:
assumes *set ws* ⊆ *G*
shows *concat ws* ∈ ⟨*G*⟩
using *assms* **by** (*induction ws*) *auto*

lemma *concat-in-hull′* [*intro*]:
assumes *ws* ∈ *lists G*
shows *concat ws* ∈ ⟨*G*⟩
using *assms* **by** (*induction ws*) *auto*

lemma *hull-concat-lists0*: *w* ∈ ⟨*G*⟩ ⟹ (∃ *ws* ∈ *lists G*. *concat ws* = *w*)
proof(*rule hull.induct*[*of* - *G*], *simp*)
show ∃ *ws*∈*lists G*. *concat ws* = ε
using *concat.simps*(*1*) *lists.Nil*[*of G*] *exI*[*of* λ *x*. *concat x* = ε, *OF concat.simps*(*1*)] **by** *blast*
show ⋀*w1 w2*. *w1* ∈ *G* ⟹ *w2* ∈ ⟨*G*⟩ ⟹ ∃ *ws*∈*lists G*. *concat ws* = *w2* ⟹ ∃ *ws*∈*lists G*. *concat ws* = *w1* · *w2*
by (*metis Cons-in-lists-iff concat.simps*(*2*))
qed

lemma *hull-concat-lists*: ⟨*G*⟩ = *concat* ‘ *lists G*
unfolding *image-iff* **using** *hull-concat-lists0* **by** *blast*

lemma *concat-tl*: *x* # *xs* ∈ *lists G* ⟹ *concat xs* ∈ ⟨*G*⟩
by (*simp add*: *hull-concat-lists*)

lemma *nemp-concat-hull*: **assumes** *us* ≠ ε **and** *us* ∈ *lists G*+
shows *concat us* ∈ ⟨*G*⟩ **and** *concat us* ≠ ε
using *assms* **by** *fastforce*+

lemma *hull-mon*: *A* ⊆ *B* ⟹ ⟨*A*⟩ ⊆ ⟨*B*⟩
proof
fix *x* **assume** *A* ⊆ *B x* ∈ ⟨*A*⟩
thus *x* ∈ ⟨*B*⟩
unfolding *image-def hull-concat-lists* **using** *sub-lists-mono*[*OF* ‹*A* ⊆ *B*›]
by *blast*
qed

lemma *emp-gen-set*: ⟨{}⟩ = {ε}
unfolding *hull-concat-lists* **by** *auto*

lemma *hull-drop-one*: ⟨*G*⟩ = ⟨*G*+⟩
proof (*intro equalityI subsetI*)
fix *x* **assume** *x* ∈ ⟨*G*⟩ **thus** *x* ∈ ⟨*G*+⟩
unfolding *hull-concat-lists* **using** *del-emp-concat lists-drop-emp′* **by** *blast*
next
fix *x* **assume** *x* ∈ ⟨*G*+⟩ **thus** *x* ∈ ⟨*G*⟩
unfolding *hull-concat-lists image-iff* **by** *auto*

qed

lemma *sing-gen-power*: $u \in \langle\{x\}\rangle \Longrightarrow \exists\, k.\ u = x^{@}k$
unfolding *hull-concat-lists* **using** *one-generated-list-power* **by** *auto*

lemma *sing-gen*: $w \in \langle\{z\}\rangle \Longrightarrow w \in z*$
using *rootI sing-gen-power* **by** *blast*

lemma *lists-gen-to-hull*: $us \in lists\ G_+ \Longrightarrow us \in lists\ \langle G\rangle_+$
using *lists-mono genset-sub* **by** *force*

lemma *rev-hull0*: $x \in rev\ `\ \langle G\rangle \Longrightarrow x \in \langle rev\ `\ G\rangle$
proof−
assume $x \in rev\ `\ \langle G\rangle$
then obtain *xs* **where** $x = rev\ (concat\ xs)$ **and** $xs \in lists\ G$
unfolding *hull-concat-lists* **by** *auto*
thus $x \in \langle rev\ `\ G\rangle$ **unfolding** *image-iff hull-concat-lists* **using** *rev-concat*[*of xs*]
by *fastforce*
qed

lemma *rev-hull1*: $x \in \langle rev\ `\ G\rangle \Longrightarrow x \in rev\ `\ \langle G\rangle$
proof−
assume $x \in \langle rev\ `\ G\rangle$
then obtain *xs* **where** $x = concat\ xs$ **and** $xs \in lists\ (rev\ `\ G)$
unfolding *hull-concat-lists* **by** *blast*
hence $rev\ x \in \langle G\rangle$
unfolding *hull-concat-lists* **using** *rev-concat* **by** *fastforce*
thus $x \in rev\ `\ \langle G\rangle$
by (*simp add*: *rev-in-conv*)
qed

lemma *rev-hull*: $rev`\langle G\rangle = \langle rev`G\rangle$
by (*simp add*: *rev-hull0 rev-hull1 set-eq-subset subsetI*)

lemma *power-in*: $x \in \langle G\rangle \Longrightarrow x^{@}k \in \langle G\rangle$
by (*induction k*, *auto*, *simp add*: *hull-closed*)

lemma *hull-closed-lists*: $us \in lists\ \langle G\rangle \Longrightarrow concat\ us \in \langle G\rangle$
proof (*induct us*, *auto*)
show $\bigwedge a\ us.\ concat\ us \in \langle G\rangle \Longrightarrow a \in \langle G\rangle \Longrightarrow \forall\, x{\in}set\ us.\ x \in \langle G\rangle \Longrightarrow a \cdot concat\ us \in \langle G\rangle$
by (*simp add*: *hull-closed*)
qed

lemma *self-gen*: $\langle\langle G\rangle\rangle = \langle G\rangle$
using *image-subsetI*[*of lists* $\langle G\rangle$ *concat* $\langle G\rangle$, *unfolded hull-concat-lists*[*of* $\langle G\rangle$, *symmetric*],
THEN subset-antisym[*OF - genset-sub*[*of* $\langle G\rangle$]]] *hull-closed-lists*[*of - G*] **by** *blast*

Intersection of hulls is a hull.

lemma *hulls-inter*: $\langle \bigcap \{\langle G\rangle \mid G.\ G \in S\}\rangle = \bigcap \{\langle G\rangle \mid G.\ G \in S\}$
proof
{fix G **assume** $G \in S$
hence $\langle \bigcap \{\langle G\rangle \mid G.\ G \in S\}\rangle \subseteq \langle G\rangle$
using *Inter-lower*[*of* $\langle G\rangle$ $\{\langle G\rangle \mid G.\ G \in S\}$] *mem-Collect-eq*[*of* $\langle G\rangle$ λ A. $\exists$ G. $G \in S \wedge A = \langle G\rangle$]
hull-mon[*of* $\bigcap \{\langle G\rangle \mid G.\ G \in S\}$ $\langle G\rangle$] **unfolding** *self-gen* **by** *auto***}**
thus $\langle \bigcap \{\langle G\rangle \mid G.\ G \in S\}\rangle \subseteq \bigcap \{\langle G\rangle \mid G.\ G \in S\}$ **by** *blast*
next
show $\bigcap \{\langle G\rangle \mid G.\ G \in S\} \subseteq \langle \bigcap \{\langle G\rangle \mid G.\ G \in S\}\rangle$
by (*simp add*: *genset-sub*)
qed

3.2 Factorization into generators

We define a decomposition (or a factorization) of a into elements of a given generating set. Such a decomposition is well defined only if the decomposed word is an element of the hull. Even int that case, however, the decomposition need not be unique.

fun *decompose* :: *'a list set* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list list* (*Dec - -* [*51,51*] *64*) **where**
decompose G u = (*SOME us*. $us \in lists\ G_+ \wedge concat\ us = u$)

lemma *dec-ex*: **assumes** $u \in \langle G\rangle$ **shows** $\exists$ *us*. ($us \in lists\ G_+ \wedge concat\ us = u$)
using *assms* **unfolding** *image-def hull-concat-lists*[*of G*] *mem-Collect-eq*
using *del-emp-concat lists-drop-emp'* **by** *metis*

lemma *decI'*: $u \in \langle G\rangle \Longrightarrow (Dec\ G\ u) \in lists\ G_+$
unfolding *decompose.simps* **using** *someI-ex*[*OF dec-ex*] **by** *blast*

lemma *decI*: $u \in \langle G\rangle \Longrightarrow concat\ (Dec\ G\ u) = u$
unfolding *decompose.simps* **using** *someI-ex*[*OF dec-ex*] **by** *blast*

lemma *dec-emp*: $Dec\ G\ \varepsilon = \varepsilon$
proof–
have *ex*: $\varepsilon \in lists\ G_+ \wedge concat\ \varepsilon = \varepsilon$
by *simp*
have *all*: ($us \in lists\ G_+ \wedge concat\ us = \varepsilon$) $\Longrightarrow$ $us = \varepsilon$ **for** *us*
using *emp-concat-emp* **by** *auto*
show *?thesis*
unfolding *decompose.simps*
using *all*[*OF someI*[*of* λ *x*. $x \in lists\ G_+ \wedge concat\ x = \varepsilon$, *OF ex*]].
qed

lemma *dec-nemp*: $u \in \langle G\rangle_+ \Longrightarrow\ Dec\ G\ u \neq \varepsilon$
using *decI*[*of u G*] **by** *force*

lemma *dec-nemp'*: $u \neq \varepsilon \Longrightarrow u \in \langle G \rangle \Longrightarrow Dec\ G\ u \neq \varepsilon$
using *dec-nemp* **by** *blast*

lemma *dec-dom'*: $u \in \langle G \rangle \Longrightarrow Dec\ G\ u \in lists\ G$
using *decI'* **by** *auto*

lemma *dec-hd*: **assumes** $u \neq \varepsilon$ $u \in \langle G \rangle$ **shows** $hd\ (Dec\ G\ u) \in G$
using *dec-nemp'*[*OF assms*] *dec-dom'*[*OF* ‹$u \in \langle G \rangle$›] *lists-hd*[*of Dec G u G*] **by** *blast*

lemma *non-gen-dec*: $u \in \langle G \rangle \Longrightarrow u \notin G \Longrightarrow Dec\ G\ u\ \neq [u]$
using *decI'* *Cons-in-lists-iff* **by** *fastforce*

3.2.1 Refinement into a specific decomposition

We extend the decomposition to lists of words. This can be seen as a refinement of a previous decomposition of some word.

fun *refine* :: *'a list set* ⇒ *'a list list* ⇒ *'a list list* (*Ref - -* [*51,51*] *65*) **where**
refine G us = *concat*(*map* (*decompose G*) *us*)

lemma *ref-morph*: $us \in lists\ \langle G \rangle \Longrightarrow vs \in lists\ \langle G \rangle \Longrightarrow refine\ G\ (us \cdot vs) = refine\ G\ us \cdot refine\ G\ vs$
using *refine.simps* **by** *simp*

lemma *ref-morph-plus*: $us \in lists\ \langle G \rangle_+ \Longrightarrow vs \in lists\ \langle G \rangle_+ \Longrightarrow refine\ G\ (us \cdot vs) = refine\ G\ us \cdot refine\ G\ vs$
using *refine.simps* **by** *simp*

lemma *ref-pop-hd*: $us \neq \varepsilon \Longrightarrow us \in lists\ \langle G \rangle \Longrightarrow refine\ G\ us = decompose\ G\ (hd\ us) \cdot refine\ G\ (tl\ us)$
unfolding *refine.simps* **using** *list.simps*(*9*)[*of decompose G hd us tl us*] **by** *simp*

lemma *ref-in*: $us \in lists\ \langle G \rangle \Longrightarrow (Ref\ G\ us) \in lists\ G_+$
proof (*induction us*, *simp*)
case (*Cons a us*)
then show *?case*
using *Cons.IH Cons.prems decI'* **by** *auto*
qed

lemma *ref*: $us \in lists\ \langle G \rangle \Longrightarrow concat\ (Ref\ G\ us) = concat\ us$
proof (*induction us*, *simp*)
case (*Cons a us*)
then show *?case*
using *Cons.IH Cons.prems decI* **by** *auto*
qed

lemma *ref-gen*: $us \in lists\ B \Longrightarrow B \subseteq \langle G \rangle \Longrightarrow Ref\ G\ us \in \langle decompose\ G\ `\ B \rangle$
by (*induct us*, *auto*)

lemma *emp-ref*: **assumes** $us \in lists\ \langle G\rangle_+$ **and** $Ref\ G\ us = \varepsilon$ **shows** $us = \varepsilon$
using *emp-concat-emp*[*OF* ‹$us \in lists\ \langle G\rangle_+$›]
ref[*OF lists-drop-emp*[*OF assms*(*1*)], *unfolded* ‹$Ref\ G\ us = \varepsilon$› *concat.simps*(*1*),*symmetric*]
by *blast*

lemma *sing-ref-sing*:
assumes $us \in lists\ \langle G\rangle_+$ **and** $refine\ G\ us = [b]$
shows $us = [b]$
proof−
have $us \neq \varepsilon$
using ‹$refine\ G\ us = [b]$› **by** *auto*
have $tl\ us \in lists\ \langle G\rangle_+$ **and** $hd\ us \in \langle G\rangle_+$
using *list.collapse*[*OF* ‹$us \neq \varepsilon$›] ‹$us \in lists\ \langle G\rangle_+$› *Cons-in-lists-iff*[*of hd us tl us* $\langle G\rangle_+$]
by *auto*
have $Dec\ G\ (hd\ us) \neq \varepsilon$
using *dec-nemp*[*OF* ‹$hd\ us \in \langle G\rangle_+$›].
have $us \in lists\ \langle G\rangle$
using ‹$us \in lists\ \langle G\rangle_+$› *lists-drop-emp* **by** *auto*
have $concat\ us = b$
using ‹$us \in lists\ \langle G\rangle$› *assms*(*2*) *ref* **by** *force*
have $refine\ G\ (tl\ us) = \varepsilon$
using *ref-pop-hd*[*OF* ‹$us \neq \varepsilon$› ‹$us \in lists\ \langle G\rangle$›] **unfolding** ‹$refine\ G\ us = [b]$›
using ‹$Dec\ G\ (hd\ us) \neq \varepsilon$› *Cons-eq-append-conv*[*of b* ε (*Dec G* (*hd us*)) (*Ref G* (*tl us*))]
Cons-eq-append-conv[*of b* ε (*Dec G* (*hd us*)) (*Ref G* (*tl us*))] *append-is-Nil-conv*[*of* - (*Ref G* (*tl us*))]
by *blast*
from *emp-ref*[*OF* ‹$tl\ us \in lists\ \langle G\rangle_+$› *this*, *symmetric*]
have $\varepsilon = tl\ us$.
from *this*[*unfolded Nil-tl*]
show *?thesis*
using ‹$us \neq \varepsilon$› ‹$concat\ us = b$› **by** *auto*
qed

lemma *ref-ex*: **assumes** $Q \subseteq \langle G\rangle$ **and** $us \in lists\ Q$
shows $Ref\ G\ us \in lists\ G_+$ **and** $concat\ (Ref\ G\ us) = concat\ us$
using *ref-in*[*OF sub-lists-mono*[*OF assms*]] *ref*[*OF sub-lists-mono*[*OF assms*]].

3.3 Basis

An important property of monoids of words is that they have a unique minimal generating set. Which is the set consisting of indecomposable elements.

The simple element is defined as a word which has only trivial decomposition into generators: a singleton.

function *simple-element* :: *'a list* ⇒ *'a list set* ⇒ *bool* (- ∈*B* - [*51*,*51*] *50*) **where**
simple-element b G = (*b* ∈ *G* ∧ (∀ *us*. *us* ∈ *lists G*$_+$ ∧ *concat us* = *b* ⟶ |*us*|

$= 1$))
using *prod.exhaust* **by** *auto*
termination
using *termination* **by** *blast*

lemma *simp-el-el*: $b \in_B G \Longrightarrow b \in G$
unfolding *simple-element.simps* **by** *blast*

lemma *simp-elD*: $b \in_B G \Longrightarrow us \in lists\ G_+ \Longrightarrow concat\ us = b \Longrightarrow |us| = 1$
unfolding *simple-element.simps* **by** *blast*

lemma *simp-el-sing*: **assumes** $b \in_B G$ $us \in lists\ G_+$ $concat\ us = b$ **shows** $us = [b]$
using *simp-elD*[*OF assms*] ‹*concat us* = *b*› *concat-len-one sing-word* **by** *fastforce*

lemma *nonsimp*: $us \in lists\ G_+ \Longrightarrow concat\ us \in_B G \Longrightarrow us = [concat\ us]$
using *simp-el-sing*[*of concat us G us*] **unfolding** *simple-element.simps*
by *blast*

lemma *emp-nonsimp*: $\neg\ \varepsilon \in_B G$
unfolding *simple-element.simps* **using** *list.size*(*3*) *concat.simps*(*1*) *lists.Nil*[*of* G_+]
by *fastforce*

lemma *basis-no-fact*: **assumes** $u \in \langle G \rangle$ **and** $v \in \langle G \rangle$ **and** $u \cdot v \in_B G$ **shows** $u = \varepsilon \vee v = \varepsilon$
proof−
have *eq1*: $concat\ ((Dec\ G\ u) \cdot (Dec\ G\ v)) = u \cdot v$
using *concat-morph*[*of Dec G u Dec G v*,*symmetric*]
unfolding *decI*[*OF* ‹$u \in \langle G \rangle$›] *decI*[*OF* ‹$v \in \langle G \rangle$›].
have *eq2*: $(Dec\ G\ u) \cdot (Dec\ G\ v) = [u \cdot v]$
using ‹$u \cdot v \in_B G$› *nonsimp*[*of* $(Dec\ G\ u) \cdot (Dec\ G\ v)$]
unfolding *eq1 append-in-lists-conv*[*of* $(Dec\ G\ u)$ $(Dec\ G\ v)$ G_+]
using *decI′*[*OF* ‹$u \in \langle G \rangle$›] *decI′*[*OF* ‹$v \in \langle G \rangle$›]
by (*meson append-in-lists-conv*)
have $Dec\ G\ u = \varepsilon \vee Dec\ G\ v = \varepsilon$
using *butlast-append*[*of Dec G u Dec G v*] **unfolding** *eq2 butlast.simps*(*2*)[*of* $u \cdot v$ ε]
using *Nil-is-append-conv*[*of Dec G u butlast* $(Dec\ G\ v)$] **by** *auto*
thus *?thesis*
using *decI*[*OF* ‹$u \in \langle G \rangle$›] *decI*[*OF* ‹$v \in \langle G \rangle$›]
concat.simps(*1*)
by *auto*
qed

lemma *simp-elI*:
assumes $b \in G$ **and** $b \neq \varepsilon$ **and** *all*: $\forall\ u\ v.\ u \neq \varepsilon \wedge u \in \langle G \rangle \wedge v \neq \varepsilon \wedge v \in \langle G \rangle \longrightarrow u \cdot v \neq b$
shows $b \in_B G$

unfolding *simple-element.simps*
proof(*simp add*: ‹$b \in G$›, *standard*, *standard*, *elim conjE*)
fix *us* **assume** $us \in lists\ G_+$ *concat us* $= b$
hence $us \neq \varepsilon$ **using** ‹$b \neq \varepsilon$› *concat.simps*(*1*) **by** *blast*
hence *hd us* $\in \langle G \rangle$ **and** *hd us* $\neq \varepsilon$
using ‹$us \in lists\ G_+$› *lists-hd gen-in* **by** *auto*
have *tl us* $= \varepsilon$
proof(*rule ccontr*)
assume *tl us* $\neq \varepsilon$
from *nemp-concat-hull*[*of tl us*, *OF this tl-lists*[*OF* ‹$us \in lists\ G_+$›]]
show *False*
using *all* ‹*hd us* $\neq \varepsilon$› ‹*hd us* $\in \langle G \rangle$› *concat.simps*(*2*)[*of hd us tl us*, *symmetric*]
unfolding *list.collapse*[*OF* ‹$us \neq \varepsilon$›] ‹*concat us* $= b$›
by *blast*
qed
hence $|us| = 1$
using ‹*concat us* $= b$› *assms*(*2*) *long-list-tl nonsing-concat-len* **by** *blast*
thus $|us| =$ *Suc 0*
by (*simp add*: ‹$b \in G$›)
qed

lemma *simp-el-indecomp*:
assumes $b \in_B G$
shows $b \in G$ **and** $b \neq \varepsilon$ **and** $\forall\ u\ v.\ u \neq \varepsilon \wedge u \in \langle G \rangle \wedge v \neq \varepsilon \wedge v \in \langle G \rangle \longrightarrow u \cdot v \neq b$
using *assms basis-no-fact emp-nonsimp simple-element.simps* **by** *blast*+

We are ready to define the *basis* as the set of all simple elements.

fun *basis* :: *'a list set* $\Rightarrow$ *'a list set* ($\mathfrak{B}$ - [*51*]) **where**
basisdef: *basis* $G = \{x.\ x \in_B G\}$

lemma *basisI*: $x \in_B G \Longrightarrow x \in \mathfrak{B}\ G$
by *simp*

lemma *basisD*: $x \in \mathfrak{B}\ G \Longrightarrow x \in_B G$
by *simp*

lemma *emp-not-basis*: $x \in \mathfrak{B}\ G \Longrightarrow x \neq \varepsilon$
using *basisD emp-nonsimp* **by** *blast*

lemma *basis-sub*: $\mathfrak{B}\ G \subseteq G$
using *basisdef* **by** *simp*

lemma *basis-drop-emp*: $(\mathfrak{B}\ G)_+ = \mathfrak{B}\ G$
using *emp-not-basis* **by** *blast*

lemma *simp-el-hull'*: **assumes** $b \in_B \langle G \rangle$ **shows** $b \in_B G$
proof−
have *all*: $\forall\, us.\ us \in lists\ G_+ \wedge concat\ us = b \longrightarrow |us| = 1$

using *assms lists-gen-to-hull* **unfolding** *simple-element.simps* **by** *metis*
have $b \in \langle G \rangle$
using *assms simp-elD* **by** *auto*
obtain *bs* **where** $bs \in lists\ G_+$ **and** $concat\ bs = b$
using *dec-ex*[*OF* ‹$b \in \langle G \rangle$›] **by** *blast*
have $b \in G$
using *lists-drop-emp*[*OF* ‹$bs \in lists\ G_+$›]
lists-gen-to-hull[*OF* ‹$bs \in lists\ G_+$›, *THEN nonsimp*[*of bs* $\langle G \rangle$]],
unfolded ‹$concat\ bs = b$›, *OF* ‹$b \in_B \langle G \rangle$›] **by** *simp*
thus $b \in_B G$
by (*simp add*: *all*)
qed

lemma *simp-el-hull*: **assumes** $b \in_B G$ **shows** $b \in_B \langle G \rangle$
using *simp-elI*[*of b* $\langle G \rangle$] **unfolding** *self-gen*
using *assms gen-in simp-el-indecomp*[*OF* ‹$b \in_B G$›] **by** *auto*

lemma *concat-tl-basis*: $x \# xs \in lists\ \mathfrak{B}\ G \Longrightarrow concat\ xs \in \langle G \rangle$
unfolding *hull-concat-lists* **by** *auto*

The basis generates the hull

lemma *set-concat-len*: **assumes** $us \in lists\ G_+\ 1 < |us|\ u \in set\ us$ **shows** $|u| < |concat\ us|$
proof−
obtain $x\ y$ **where** $us = x \cdot [u] \cdot y$ **and** $x \cdot y \neq \varepsilon$
using *split-list-long*[*OF* ‹$1 < |us|$› ‹$u \in set\ us$›].
hence $x \cdot y \in lists\ G_+$
using ‹$us \in lists\ G_+$› **by** *auto*
hence $|concat\ (x \cdot y)| \neq 0$
using ‹$x \cdot y \neq \varepsilon$› *in-lists-conv-set* **by** *force*
hence $|concat\ us| = |u| + |concat\ x| + |concat\ y|$
using *length-append* ‹$us = x \cdot [u] \cdot y$› **by** *simp*
thus *?thesis*
using ‹$|concat\ (x \cdot y)| \neq 0$› **by** *auto*
qed

lemma *non-simp-dec*: **assumes** $w \notin \mathfrak{B}\ G\ w \neq \varepsilon\ w \in G$
obtains *us* **where** $us \in lists\ G_+\ 1 < |us|\ concat\ us = w$
using ‹$w \neq \varepsilon$› ‹$w \in G$› ‹$w \notin \mathfrak{B}\ G$› *nonsing-concat-len* *basisI*[*of w G*, *unfolded simple-element.simps*]
by *blast*

lemma *basis-gen*: $w \in G \longrightarrow w \in \langle \mathfrak{B}\ G \rangle$
proof (*induct length w arbitrary*: *w rule*: *less-induct*)
case *less*
{assume $w \notin \mathfrak{B}\ G\ w \neq \varepsilon\ w \in G$
obtain *us* **where** $us \in lists\ G_+\ 1 < |us|\ concat\ us = w$
using *non-simp-dec*[*OF* ‹$w \notin \mathfrak{B}\ G$› ‹$w \neq \varepsilon$› ‹$w \in G$›] **by** *blast*
have $u \in set\ us \Longrightarrow u \in \langle \mathfrak{B}\ G \rangle$ **for** *u*

```
    using lists-drop-emp[OF ‹us ∈ lists G₊›]
     set-concat-len[OF ‹us ∈ lists G₊› ‹1 < |us|›, THEN less[unfolded ‹concat us
= w›[symmetric], of u]]
     unfolding in-lists-conv-set[of us G] by blast
   from subsetI[of set us, OF this]
   have ?case
   using concat-in-hull[of us ⟨𝔅 G⟩, unfolded self-gen ‹concat us = w›] by blast
  }
 thus ?case
  by auto
qed

theorem basis-gen-hull: ⟨𝔅 G⟩ = ⟨G⟩
proof
 show ⟨𝔅 G⟩ ⊆ ⟨G⟩
  unfolding hull-concat-lists by auto
 show ⟨G⟩ ⊆ ⟨𝔅 G⟩
 proof
  fix x show x ∈ ⟨G⟩ ⟹ x ∈ ⟨𝔅 G⟩
  proof (induct rule: hull.induct)
   show ⋀w1 w2. w1 ∈ G ⟹ w2 ∈ ⟨𝔅 G⟩ ⟹ w1 · w2 ∈ ⟨𝔅 G⟩
    using hull-closed[of - 𝔅 G] basis-gen[of - G] by blast
  qed auto
 qed
qed

lemma basis-gen-hull′: ⟨𝔅 ⟨G⟩⟩ = ⟨G⟩
 using basis-gen-hull self-gen by blast

theorem basis-of-hull: 𝔅 G = 𝔅 ⟨G⟩
proof
 show 𝔅 G ⊆ 𝔅 ⟨G⟩
  using basisD basisI simp-el-hull by blast
 show 𝔅 ⟨G⟩ ⊆ 𝔅 G
  using basisD basisI simp-el-hull′ by blast
qed
```

The basis is the smallest generating set.

```
theorem ⟨S⟩ = ⟨G⟩ ⟹ 𝔅 G ⊆ S
 by (metis basis-of-hull basis-sub)
```

An arbitrary set between basis and the hull is generating...

```
lemma gen-sets: assumes 𝔅 G ⊆ S and S ⊆ ⟨G⟩ shows ⟨S⟩ = ⟨G⟩
  using image-mono[OF lists-mono[of S ⟨G⟩], of concat, OF ‹S ⊆ ⟨G⟩›] im-
age-mono[OF lists-mono[of 𝔅 G S], of concat, OF ‹𝔅 G ⊆ S›]
 unfolding sym[OF hull-concat-lists] basis-gen-hull
 using subset-antisym[of ⟨S⟩ ⟨G⟩] self-gen by auto
```

... and has the same basis

lemma *basis-sets*: $\mathfrak{B}\ G \subseteq S \Longrightarrow S \subseteq \langle G\rangle \Longrightarrow \mathfrak{B}\ G = \mathfrak{B}\ S$
by (*metis basis-of-hull gen-sets*)

Any nonempty composed element has a decomposition into basis elements with many useful properties

lemma *non-simp-fac*: **assumes** $w \neq \varepsilon$ **and** $w \in \langle G\rangle$ **and** $w \notin \mathfrak{B}\ G$
obtains *us* **where** $1 < |us|$ **and** $us \neq \varepsilon$ **and** $us \in lists\ \mathfrak{B}\ G$ **and**
$hd\ us \neq \varepsilon$ **and** $hd\ us \in \langle G\rangle$ **and**
$concat(tl\ us) \neq \varepsilon$ **and** $concat(tl\ us) \in \langle G\rangle$ **and**
$w = hd\ us \cdot concat(tl\ us)$
proof−
obtain *us* **where** $us \in lists\ \mathfrak{B}\ G$ **and** $concat\ us = w$
using ‹$w \in \langle G\rangle$› *dec-dom'*[*of w* $\mathfrak{B}$ *G*] *decI*[*of w* $\mathfrak{B}$ *G*]
unfolding *basis-gen-hull*
by *blast*
hence $us \neq \varepsilon$
using ‹$w \neq \varepsilon$› *concat.simps*(*1*)
by *blast*
from *lists-hd*[*OF this* ‹$us \in lists\ \mathfrak{B}\ G$›, *THEN emp-not-basis*]
lists-hd[*OF this* ‹$us \in lists\ \mathfrak{B}\ G$›, *THEN gen-in*[*of hd us* $\mathfrak{B}$ *G*, *unfolded basis-gen-hull*]]
have $hd\ us \neq \varepsilon$ **and** $hd\ us \in \langle G\rangle$**.**
have $1 < |us|$
using ‹$w \notin \mathfrak{B}\ G$› *lists-hd*[*OF* ‹$us \neq \varepsilon$› ‹$us \in lists\ \mathfrak{B}\ G$›] ‹$w \neq \varepsilon$› ‹$w \in \langle G\rangle$›
concat-len-one[*of us*, *unfolded* ‹*concat us* = *w*›] *nonsing-concat-len*[*of us*, *unfolded* ‹*concat us* = *w*›] **by** *blast*
from *nemp-concat-hull*[*OF long-list-tl*[*OF this*], *of* $\mathfrak{B}$ *G*, *unfolded basis-drop-emp basis-gen-hull*, *OF tl-lists*[*OF* ‹$us \in lists\ \mathfrak{B}\ G$›]]
have $concat\ (tl\ us) \in \langle G\rangle$ **and** $concat(tl\ us) \neq \varepsilon$**.**
have $w = hd\ us \cdot concat(tl\ us)$
using ‹$us \neq \varepsilon$› ‹$us \in lists\ \mathfrak{B}\ G$› ‹*concat us* = *w*› *concat.simps*(*2*)[*of hd us tl us*] *list.collapse*[*of us*]
by *argo*
from *that*[*OF* ‹$1 < |us|$› ‹$us \neq \varepsilon$› ‹$us \in lists\ \mathfrak{B}\ G$› ‹$hd\ us \neq \varepsilon$› ‹$hd\ us \in \langle G\rangle$› ‹$concat\ (tl\ us) \neq \varepsilon$› ‹$concat\ (tl\ us) \in \langle G\rangle$› *this*]
show *thesis***.**
qed

lemma *basis-dec*: $p \in \langle G\rangle \Longrightarrow s \in \langle G\rangle \Longrightarrow p \cdot s \in \mathfrak{B}\ G \Longrightarrow p = \varepsilon \vee s = \varepsilon$
using *basis-no-fact*[*of p G s*] **by** *simp*

lemma *non-simp-fac'*: $w \notin \mathfrak{B}\ G \Longrightarrow w \neq \varepsilon \Longrightarrow w \in \langle G\rangle \Longrightarrow \exists\, us.\ us \in lists\ G_+ \wedge w = concat\ us \wedge |us| \neq 1$
by (*metis basisI concat-len-one decI' dec-dom' decI dec-nemp lists-hd nemp-elem-setI simple-element.elims*(*3*))

lemma *emp-gen-iff*: $G_+ = \{\} \longleftrightarrow \langle G\rangle = \{\varepsilon\}$
proof
assume $G_+ = \{\}$ **show** $\langle G\rangle = \{\varepsilon\}$

using *hull-drop-one*[*of* G, *unfolded* ‹$G_+ = \{\}$› *emp-gen-set*].
next
assume $\langle G \rangle = \{\varepsilon\}$ **thus** $G_+ = \{\}$ **by** *blast*
qed

lemma *emp-basis-iff*: $\mathfrak{B}\ G = \{\} \longleftrightarrow G_+ = \{\}$
using *emp-gen-iff*[*of* $\mathfrak{B}\ G$, *unfolded basis-gen-hull basis-drop-emp*, *folded emp-gen-iff*].

3.4 Code

A basis freely generating its hull is called a *code*. By definition, this means that generated elements have unique factorizations into the elements of the code.

locale *code* =
fixes $\mathcal{C}$
assumes $\mathcal{C}$*-is-code*: $xs \in lists\ \mathcal{C} \Longrightarrow ys \in lists\ \mathcal{C} \Longrightarrow concat\ xs = concat\ ys \Longrightarrow xs = ys$
begin

lemma *emp-not-in-code*: $\varepsilon \notin \mathcal{C}$
proof
assume $\varepsilon \in \mathcal{C}$
hence $[] \in lists\ \mathcal{C}$ **and** $[\varepsilon] \in lists\ \mathcal{C}$ **and** $concat\ [] = concat\ [\varepsilon]$ **and** $[] \neq [\varepsilon]$
by *simp+*
thus *False*
using $\mathcal{C}$*-is-code* **by** *blast*
qed

lemma *code-simple*: $c \in \mathcal{C} \Longrightarrow c \in_B \mathcal{C}$
unfolding *simple-element.simps*
proof
fix c **assume** $c \in \mathcal{C}$
hence $[c] \in lists\ \mathcal{C}$
by *simp*
show $\forall us.\ us \in lists\ \mathcal{C}_+ \wedge concat\ us = c \longrightarrow |us| = 1$
proof
fix *us*
{assume $us \in lists\ \mathcal{C}_+$ $concat\ us = c$
hence $us \in lists\ \mathcal{C}$ **by** *blast*
hence $us = [c]$
using ‹$concat\ us = c$› ‹$c \in \mathcal{C}$› $\mathcal{C}$*-is-code*[*of* $[c]$, *OF* ‹$[c] \in lists\ \mathcal{C}$› ‹$us \in lists\ \mathcal{C}$›] *emp-not-in-code* **by** *auto***}**
thus $us \in lists\ \mathcal{C}_+ \wedge concat\ us = c \longrightarrow |us| = 1$
using *sing-len*[*of* c] **by** *fastforce*
qed
qed

lemma *code-is-basis*: $\mathfrak{B}\ \mathcal{C} = \mathcal{C}$

using *code-simple basisdef*[*of* $\mathcal{C}$] *basis-sub* **by** *blast*

lemma *code-unique-dec*: $us \in lists\ \mathcal{C} \Longrightarrow Dec\ \mathcal{C}\ (concat\ us) = us$
using *dec-dom'*[*of concat us* $\mathcal{C}$, *THEN* $\mathcal{C}$*-is-code*, *of us*]
decI[*of concat us* $\mathcal{C}$] *hull-concat-lists*[*of* $\mathcal{C}$] *image-eqI*[*of concat us concat us lists* $\mathcal{C}$]
by *argo*

lemma *code-unique-ref*: $us \in lists\ \langle\mathcal{C}\rangle \Longrightarrow refine\ \mathcal{C}\ us = decompose\ \mathcal{C}\ (concat\ us)$
proof–
assume $us \in lists\ \langle\mathcal{C}\rangle$
hence $concat\ (refine\ \mathcal{C}\ us) = concat\ us$
using *ref* **by** *fastforce*
hence *eq*: $concat\ (refine\ \mathcal{C}\ us) = concat\ (decompose\ \mathcal{C}\ (concat\ us))$
using *decI*[*OF hull-closed-lists*[*OF* ‹$us \in lists\ \langle\mathcal{C}\rangle$›]] **by** *auto*
have *dec*: $Dec\ \mathcal{C}\ (concat\ us) \in lists\ \mathcal{C}$
using ‹$us \in lists\ \langle\mathcal{C}\rangle$› *dec-dom' hull-closed-lists* **by** *blast*
have *ref*: $Ref\ \mathcal{C}\ us \in lists\ \mathcal{C}$
using *lists-drop-emp*[*OF ref-in*[*OF* ‹$us \in lists\ \langle\mathcal{C}\rangle$›]].
show *?thesis*
using $\mathcal{C}$*-is-code*[*OF ref dec eq*].
qed

lemma *code-dec-morph*: **assumes** $x \in \langle\mathcal{C}\rangle\ y \in \langle\mathcal{C}\rangle$
shows $(Dec\ \mathcal{C}\ x) \cdot (Dec\ \mathcal{C}\ y) = Dec\ \mathcal{C}\ (x{\cdot}y)$
proof–
have *eq*: $(Dec\ \mathcal{C}\ x) \cdot (Dec\ \mathcal{C}\ y) = Dec\ \mathcal{C}\ (concat\ ((Dec\ \mathcal{C}\ x) \cdot (Dec\ \mathcal{C}\ y)))$
using *dec-dom'*[*OF* ‹$x \in \langle\mathcal{C}\rangle$›] *dec-dom'*[*OF* ‹$y \in \langle\mathcal{C}\rangle$›]
code.code-unique-dec[*OF code-axioms*, *of* $(Dec\ \mathcal{C}\ x) \cdot (Dec\ \mathcal{C}\ y)$, *unfolded append-in-lists-conv*, *symmetric*]
by *blast*
moreover have $concat\ ((Dec\ \mathcal{C}\ x) \cdot (Dec\ \mathcal{C}\ y)) = (x \cdot y)$
using *concat-morph*[*of Dec* $\mathcal{C}$ *x Dec* $\mathcal{C}$ *y*, *symmetric*]
unfolding *decI*[*OF* ‹$x \in \langle\mathcal{C}\rangle$›] *decI*[*OF* ‹$y \in \langle\mathcal{C}\rangle$›].
ultimately show $(Dec\ \mathcal{C}\ x) \cdot (Dec\ \mathcal{C}\ y) = Dec\ \mathcal{C}\ (x{\cdot}y)$
by *argo*
qed

lemma *code-el-dec*: $c \in \mathcal{C} \Longrightarrow decompose\ \mathcal{C}\ c = [c]$
using *code-unique-dec*[*of* $[c]$] **by** *auto*

lemma *code-ref-list*: $us \in lists\ \mathcal{C} \Longrightarrow refine\ \mathcal{C}\ us = us$
proof (*induct us*)
case (*Cons a us*)
then show *?case* **using** *code-el-dec*
by *simp*
qed *simp*

lemma *code-ref-gen*: **assumes** $G \subseteq \langle\mathcal{C}\rangle\ u \in \langle G\rangle$

shows *Dec* $\mathcal{C}$ $u \in \langle$*decompose* $\mathcal{C}$ ‘ $G\rangle$
proof–
have *refine* $\mathcal{C}$ (*Dec* G u) = *Dec* $\mathcal{C}$ u
using *dec-dom'*[*OF* ‹$u \in \langle G\rangle$›] ‹$G \subseteq \langle\mathcal{C}\rangle$› *code-unique-ref*[*of Dec G u*, *unfolded decI*[*OF* ‹$u \in \langle G\rangle$›]] **by** *blast*
from *ref-gen*[*of Dec G u G*, *OF dec-dom'*[*OF* ‹$u \in \langle G\rangle$›], *of* $\mathcal{C}$, *unfolded this*, *OF* ‹$G \subseteq \langle\mathcal{C}\rangle$›]
show *?thesis***.**
qed

end — end context code

3.5 Binary code

We pay a special attention to two element codes. In particular, we show that two words form a code if and only if they do not commute. This means that two words either commute, or do not satisfy any nontrivial relation.

locale *binary-code* =
fixes u_0 u_1
assumes *non-comm*: $u_0 \cdot u_1 \neq u_1 \cdot u_0$

begin

lemma *bin-fst-nemp*: $u_0 \neq \varepsilon$
using *non-comm* **by** *auto*

A crucial property of two element codes is the constant decoding delay given by the word α, which is a prefix of any generating word (sufficiently long), while the letter immediately after this common prefix indicates the first element of the decomposition.

definition α **where** *bin-lcp-def* [*simp*]: $\alpha = u_0{\cdot}u_1 \wedge_p u_1{\cdot}u_0$
definition c_0 **where** *fst-mismatch-def*: $c_0 = (u_0{\cdot}u_1)!|\alpha|$
definition c_1 **where** *snd-mismatch-def*: $c_1 = (u_1{\cdot}u_0)!|\alpha|$

lemma *bin-mismatch-neq*: $c_0 \neq c_1$
unfolding *fst-mismatch-def snd-mismatch-def bin-lcp-def*
using *non-comm lcp-mismatch' pref-comp-eq*[*of* $u_0 \cdot u_1$ $u_1 \cdot u_0$, *OF - swap-len*]
unfolding *prefix-comparable-def*
by *blast*

lemma *bin-lcp-pref-fst-snd*: $\alpha \leq p\ u_0 \cdot u_1$ **and** *bin-lcp-pref-snd-fst*: $\alpha \leq p\ u_1 \cdot u_0$
unfolding *bin-lcp-def* **using** *longest-common-prefix-prefix1 longest-common-prefix-prefix2***.**

lemma *bin-lcp-short*: $|\alpha| < |u_0| + |u_1|$
proof–
have ¬ $u_0{\cdot}u_1 \leq p\ u_1{\cdot}u_0$
using *comm-ruler non-comm* **by** *blast*

from *lcp-len'*[*OF this*, *folded bin-lcp-def*, *unfolded length-append*]
show $|\alpha| < |u_0| + |u_1|$.
qed

lemma *bin-lcp-fst-mismatch-pref*: $\alpha \cdot [c_0] \leq p\ u_0 \cdot \alpha$
proof−
have $\alpha \cdot [c_0] \leq p\ u_0 \cdot u_1$
using *append-one-prefix*[*of* $\alpha\ u_0 \cdot u_1$, *folded fst-mismatch-def*, *OF bin-lcp-pref-fst-snd*, *unfolded length-append*, *OF bin-lcp-short*].
hence $\alpha \cdot [c_0] \leq p\ u_0 \cdot (u_1 \cdot u_0)$
using *pref-prolong* **by** *blast*
from *pref-prod-pref-short*[*OF this bin-lcp-pref-snd-fst*, *unfolded length-append sing-len*]
show $\alpha \cdot [c_0] \leq p\ u_0 \cdot \alpha$
using *nemp-len*[*OF bin-fst-nemp*] **by** *linarith*
qed

lemma *not-fst-snd-pref*: $\neg\ u_0 \cdot u_1 \leq p\ \alpha$
using *bin-lcp-short*[*folded length-append*[*of* $u_0\ u_1$]] *prefix-order.antisym*[*OF bin-lcp-pref-fst-snd*]
by *fastforce*

lemma *bin-lcp-fst-mismatch-pref'*: $\alpha \cdot [c_0] \leq p\ u_0 \cdot u_1$
using *strict-prefixI*[*OF bin-lcp-pref-fst-snd*, *THEN add-nth-pref*, *folded fst-mismatch-def*] *not-fst-snd-pref*
self-pref[*of* α] **by** *fastforce*

interpretation *symcode*: *binary-code* $u_1\ u_0$
rewrites *symcode.*$c_0 = c_1$ **and** *symcode.*$c_1 = c_0$ **and** *symcode.*$\alpha = \alpha$
proof−
show *binary-code* $u_1\ u_0$
unfolding *binary-code-def* **using** *non-comm* **by** *simp*
show *binary-code.*$\alpha\ u_1\ u_0 = \alpha$
by (*simp add*: ‹*binary-code* $u_1\ u_0$› *binary-code.bin-lcp-def lcp-sym*)
show *binary-code.*$c_0\ u_1\ u_0 = c_1$
by (*simp add*: ‹*binary-code* $u_1\ u_0$› ‹*binary-code.*$\alpha\ u_1\ u_0 = \alpha$› *binary-code.fst-mismatch-def snd-mismatch-def*)
show *binary-code.*$c_1\ u_1\ u_0 = c_0$
by (*simp add*: ‹*binary-code* $u_1\ u_0$› ‹*binary-code.*$\alpha\ u_1\ u_0 = \alpha$› *binary-code.snd-mismatch-def fst-mismatch-def*)
qed

lemmas *bin-snd-nemp* = *symcode.bin-fst-nemp* **and**
bin-snd-mismatch = *symcode.bin-lcp-fst-mismatch-pref*

lemma *bin-lcp-fst-lcp*: $\alpha \leq p\ u_0 \cdot \alpha$ **and** *bin-lcp-snd-lcp*: $\alpha \leq p\ u_1 \cdot \alpha$
using *bin-lcp-fst-mismatch-pref* *bin-snd-mismatch* **by** *auto*

lemma *bin-all-nemp*: *ws* $\in$ *lists* $\{u_0,u_1\} \Longrightarrow$ *concat ws* $= \varepsilon \Longrightarrow$ *ws* $= \varepsilon$
using *bin-fst-nemp bin-snd-nemp* **by**(*induct ws*, *simp*, *auto*)

lemma *bin-lcp-all-lcp*: *ws* ∈ *lists* {u_0,u_1} ⟹ $\alpha \leq_p$ *concat ws* · α
proof(*induct ws rule*: *rev-induct*, *simp*)
case (*snoc x xs*)
have *x-or*: $x = u_0 \vee x = u_1$
using ‹*xs* · [*x*] ∈ *lists* {u_0, u_1}› **by** *simp*
have *xs* ∈ *lists* {u_0, u_1}
using ‹*xs* · [*x*] ∈ *lists* {u_0, u_1}› **by** *auto*
from *pref-prolong*[*OF snoc.hyps*, *OF this*, *of x*·α, *unfolded lassoc*]
show *?case*
using *bin-lcp-fst-lcp bin-lcp-snd-lcp disjE*[*OF x-or*] **by** *auto*
qed

lemma *bin-code-alpha*: **assumes** *us* ∈ *lists* {u_0,u_1} **and** *vs* ∈ *lists* {u_0,u_1} **and** *hd us* ≠ *hd vs*
shows *concat us* · $\alpha \wedge_p$ *concat vs* · $\alpha = \alpha$
using *assms*
proof (*induct us vs rule*: *list-induct2′*, *simp*)
case (*2 x xs*)
show *?case*
using *bin-lcp-all-lcp*[*OF* ‹*x* # *xs* ∈ *lists* {u_0, u_1}›, *folded lcp-pref-conv*, *unfolded lcp-sym*[*of* α]] **by** *simp*
next
case (*3 y ys*)
show *?case*
using *bin-lcp-all-lcp*[*OF* ‹*y* # *ys* ∈ *lists* {u_0, u_1}›, *folded lcp-pref-conv*] **by** *simp*
next
case (*4 x xs y ys*)
interpret *i*: *binary-code x y*
using *4.prems*(*1*) *4.prems*(*2*) *4.prems*(*3*) *non-comm binary-code.intro* **by** *auto*
have *alph*: {u_0,u_1} = {*x*,*y*}
using *4.prems*(*1*) *4.prems*(*2*) *4.prems*(*3*) **by** *auto*
from *disjE*[*OF this*[*unfolded doubleton-eq-iff*]]
have α*id*: *i*.α = α
unfolding *bin-lcp-def i.bin-lcp-def* **using** *lcp-sym* **by** *auto*
have *c0*: *i*.α · [*i*.c_0] ≤*p x* · *concat xs* · *i*.α
using *i.bin-lcp-all-lcp*[*of xs*] ‹*x* # *xs* ∈ *lists* {u_0, u_1}›[*unfolded Cons-in-lists-iff alph*]
pref-prolong[*OF i.bin-lcp-fst-mismatch-pref*] **by** *blast*
have *c1*: *i*.α · [*i*.c_1] ≤*p y* · *concat ys* · *i*.α
using *i.bin-lcp-all-lcp*[*of ys*] ‹*y* # *ys* ∈ *lists* {u_0, u_1}›[*unfolded Cons-in-lists-iff alph*]
pref-prolong[*OF i.bin-snd-mismatch*] **by** *blast*
have *i*.α·[*i*.c_0] $\wedge_p$ *i*.α·[*i*.c_1] = *i*.α
by (*simp add*: *i.bin-mismatch-neq lcp-first-mismatch′*)
from *lcp-rulers*[*OF c0 c1*, *unfolded this*, *unfolded* α*id*]
show *?case*
unfolding *concat.simps*(*2*) *rassoc pref-cancel-conv* **using** *i.bin-mismatch-neq* **by** *simp*

qed

theorem *bin-code*: **assumes** *us* ∈ *lists* {u_0,u_1} **and** *vs* ∈ *lists* {u_0,u_1} **and** *concat us* = *concat vs*
shows *us* = *vs*
using *assms*
proof (*induct us vs rule*: *list-induct2′*, *simp*)
case (*2 x xs*)
then show *?case*
using *bin-fst-nemp bin-snd-nemp* **by** *auto*
next
case (*3 y ys*)
then show *?case*
using *bin-fst-nemp bin-snd-nemp* **by** *auto*
next
case (*4 x xs y ys*)
then show *?case*
proof(*cases x* = *y*)
assume *x* = *y* **thus** *x* # *xs* = *y* # *ys*
using *4.hyps* ‹*concat* (*x* # *xs*) = *concat* (*y* # *ys*)›[*unfolded concat.simps*(*2*) ‹*x* = *y*›, *unfolded cancel*]
‹*y* # *ys* ∈ *lists* {u_0, u_1}›[*unfolded Cons-in-lists-iff*] ‹*x* # *xs* ∈ *lists* {u_0, u_1}›[*unfolded Cons-in-lists-iff*]
by *simp*
next
assume *x* ≠ *y*
have *concat*(*y* # *ys*) = ε
using *bin-code-alpha*[*OF* ‹*x* # *xs* ∈ *lists* {u_0, u_1}› ‹*y* # *ys* ∈ *lists* {u_0, u_1}›, *unfolded list.sel*(*1*) ‹*concat* (*x* # *xs*) = *concat* (*y* # *ys*)›, *OF* ‹*x* ≠ *y*›]
by *simp*
from *bin-all-nemp*[*OF* ‹*y* # *ys* ∈ *lists* {u_0, u_1}› *this*]
have *False* **by** *simp*
thus *x* # *xs* = *y* # *ys* **by** *blast*
qed
qed

end

lemmas *no-comm-bin-code* = *binary-code.bin-code*[*unfolded binary-code-def*]

theorem *bin-code-code*: **assumes** *u* · *v* ≠ *v* · *u* **shows** *code* {*u*, *v*}
unfolding *code-def* **using** *no-comm-bin-code*[*OF assms*] **by** *blast*

3.6 Free hull

While not every set G of generators is a code, there is a unique minimal free monoid containing it, called the *free hull* of G. It can be defined inductively using the property known as the *stability condition*.

inductive-set *free-hull* :: *'a list set* ⇒ *'a list set* (⟨-⟩$_F$)
  for G **where**
    $\varepsilon \in \langle G\rangle_F$
  | *free-gen-in*: $w \in G \Longrightarrow w \in \langle G\rangle_F$
  | $w1 \in \langle G\rangle_F \Longrightarrow w2 \in \langle G\rangle_F \Longrightarrow w1 \cdot w2 \in \langle G\rangle_F$
  | $p \in \langle G\rangle_F \Longrightarrow q \in \langle G\rangle_F \Longrightarrow p \cdot w \in \langle G\rangle_F \Longrightarrow w \cdot q \in \langle G\rangle_F \Longrightarrow w \in \langle G\rangle_F$ —
the stability condition

lemmas [*intro*] = *free-hull.intros*

The defined set indeed is a hull.

lemma *free-hull-hull*: $\langle\langle G\rangle_F\rangle = \langle G\rangle_F$
proof
  show $\langle G\rangle_F \subseteq \langle\langle G\rangle_F\rangle$
    by (*simp add*: *genset-sub*)
  show $\langle\langle G\rangle_F\rangle \subseteq \langle G\rangle_F$
  proof
    fix x **assume** $x \in \langle\langle G\rangle_F\rangle$
    thus $x \in \langle G\rangle_F$
    proof (*rule hull.induct*)
      show $\varepsilon \in \langle G\rangle_F$
        by (*simp add*: *free-hull.intros*(*1*))
      show $\bigwedge w1\ w2.\ w1 \in \langle G\rangle_F \Longrightarrow w2 \in \langle\langle G\rangle_F\rangle \Longrightarrow w2 \in \langle G\rangle_F \Longrightarrow w1 \cdot w2 \in \langle G\rangle_F$
        by (*simp add*: *free-hull.intros*(*3*))
    qed
  qed
qed

The free hull is always (non-strictly) larger than the hull.

lemma *hull-in-free-hull*: $\langle G\rangle \subseteq \langle G\rangle_F$
proof
  fix x **assume** $x \in \langle G\rangle$
  then show $x \in \langle G\rangle_F$
    using *free-hull.intros*(*3*)
      hull-induct[*of* x G λ x. $x \in \langle G\rangle_F$, *OF* ‹$x \in \langle G\rangle$› *free-hull.intros*(*1*)[*of* G] *free-hull.intros*(*2*)]
    by *auto*
qed

On the other hand, it can be proved that the *free basis*, defined as the basis of the free hull, has a (non-strictly) smaller cardinality than the ordinary basis.

definition *free-basis* :: *'a list set* ⇒ *'a list set* ($\mathfrak{B}_F$ - [*54*] *55*)
  where *free-basis* $G \equiv \mathfrak{B}\ \langle G\rangle_F$

lemma *basis-gen-hull-free*: $\langle \mathfrak{B}_F\ G\rangle = \langle G\rangle_F$
  unfolding *free-basis-def* **using** *basis-gen-hull free-hull-hull* **by** *blast*

lemma *genset-sub-free*: $G \subseteq \langle G \rangle_F$
by (*simp add*: *free-hull.free-gen-in subsetI*)

We have developed two points of view on freeness:

- being a free hull, that is, to satisfy the stability condition;
- being generated by a code.

We now show their equivalence

First, basis of a free hull is a code.

lemma *free-basis-code*: *code* ($\mathfrak{B}_F$ G)
proof
fix *xs ys*
show *xs* ∈ *lists* ($\mathfrak{B}_F$ G) ⟹ *ys* ∈ *lists* ($\mathfrak{B}_F$ G) ⟹ *concat xs* = *concat ys* ⟹ *xs* = *ys*
proof(*induction xs ys rule*: *list-induct2′*, *simp*)
case (*2 x xs*)
show *?case*
using *listsE*[*OF* ‹*x* # *xs* ∈ *lists* ($\mathfrak{B}_F$ G)›, *of x* ∈ $\mathfrak{B}_F$ G, *unfolded free-basis-def*, *THEN emp-not-basis*]
concat.simps(*2*)[*of x xs*, *unfolded* ‹*concat* (*x* # *xs*) = *concat* ε›[*unfolded concat.simps*(*1*)], *symmetric*, *unfolded append-is-Nil-conv*[*of x concat xs*]]
by *blast*
next
case (*3 y ys*)
show *?case*
using *listsE*[*OF* ‹*y* # *ys* ∈ *lists* ($\mathfrak{B}_F$ G)›, *of y* ∈ $\mathfrak{B}_F$ G, *unfolded free-basis-def*, *THEN emp-not-basis*]
concat.simps(*2*)[*of y ys*, *unfolded* ‹*concat* ε = *concat* (*y* # *ys*)›[*unfolded concat.simps*(*1*),*symmetric*],*symmetric*, *unfolded append-is-Nil-conv*[*of y concat ys*]]
by *blast*
next
case (*4 x xs y ys*)
have |*x*| = |*y*|
proof(*rule ccontr*)
assume |*x*| ≠ |*y*|
have *x* · *concat xs* = *y* · *concat ys*
using ‹*concat* (*x* # *xs*) = *concat* (*y* # *ys*)› **by** *simp*
then obtain *t* **where** *or*: *x* = *y* · *t* ∧ *t* · *concat xs* = *concat ys* ∨ *x* · *t* = *y* ∧ *concat xs* = *t* · *concat ys*
using *append-eq-append-conv2*[*of x concat xs y concat ys*] **by** *blast*
hence *t* ≠ ε
using ‹|*x*| ≠ |*y*|› **by** *auto*
have *x* ∈ $\mathfrak{B}_F$ G **and** *y* ∈ $\mathfrak{B}_F$ G
using *listsE*[*OF* ‹*x* # *xs* ∈ *lists* ($\mathfrak{B}_F$ G)›, *of x* ∈ $\mathfrak{B}_F$ G] *listsE*[*OF* ‹*y* # *ys* ∈ *lists* ($\mathfrak{B}_F$ G)›, *of y* ∈ $\mathfrak{B}_F$ G] **by** *blast*+

hence $x \neq \varepsilon$ **and** $y \neq \varepsilon$
unfolding *free-basis-def* **using** *emp-not-basis* **by** *blast+*
have $x \in \langle G \rangle_F$ **and** $y \in \langle G \rangle_F$
using *basis-sub*[*of* $\langle G \rangle_F$, *unfolded free-basis-def*[*symmetric*]] ‹$x \# xs \in lists\ (\mathfrak{B}_F\ G)$›
‹$y \# ys \in lists\ (\mathfrak{B}_F\ G)$› **by** *auto*
have *concat xs* $\in \langle G \rangle_F$ **and** *concat ys* $\in \langle G \rangle_F$
using *concat-tl-basis*[*OF* ‹$x \# xs \in lists\ (\mathfrak{B}_F\ G)$›[*unfolded free-basis-def*]]
concat-tl-basis[*OF* ‹$y \# ys \in lists\ (\mathfrak{B}_F\ G)$›[*unfolded free-basis-def*]]
unfolding *free-hull-hull*.
have $t \in \langle G \rangle_F$
using *or free-hull.intros*(*4*) ‹$x \in \langle G \rangle_F$› ‹$y \in \langle G \rangle_F$› ‹*concat xs* $\in \langle G \rangle_F$› ‹*concat ys* $\in \langle G \rangle_F$› **by** *metis*
thus *False*
using *or basis-dec*[*of x* $\langle G \rangle_F$ *t*, *unfolded free-hull-hull*, *OF* ‹$x \in \langle G \rangle_F$› ‹$t \in \langle G \rangle_F$›]
basis-dec[*of y* $\langle G \rangle_F$ *t*, *unfolded free-hull-hull*, *OF* ‹$y \in \langle G \rangle_F$› ‹$t \in \langle G \rangle_F$›]
using ‹$t \neq \varepsilon$› ‹$x \neq \varepsilon$› ‹$y \neq \varepsilon$› ‹$x \in \mathfrak{B}_F\ G$› ‹$y \in \mathfrak{B}_F\ G$› **unfolding** *free-basis-def*
by *auto*
qed
thus $x \# xs = y \# ys$
using *4.IH* ‹$x \# xs \in lists\ (\mathfrak{B}_F\ G)$› ‹$y \# ys \in lists\ (\mathfrak{B}_F\ G)$› ‹*concat* $(x \# xs) = concat\ (y \# ys)$›
by *auto*
next
qed
qed

Second, a code generates its free hull.

lemma *code-gen-free-hull*: *code* $C \Longrightarrow \langle C \rangle_F = \langle C \rangle$
proof
assume *code C*
show $\langle C \rangle \subseteq \langle C \rangle_F$
using *hull-mon*[*of C* $\langle C \rangle_F$]
free-hull.free-gen-in[*of - C*] *subsetI*[*of C* $\langle C \rangle_F$]
unfolding *free-hull-hull*[*of C*] **by** *auto*
show $\langle C \rangle_F \subseteq \langle C \rangle$
proof
fix *x* **assume** $x \in \langle C \rangle_F$
have $\varepsilon \in \langle C \rangle$
by (*simp add*: *hull.intros*(*1*))
show $x \in \langle C \rangle$
proof(*rule free-hull.induct*[*of x C*],*simp add*: ‹$x \in \langle C \rangle_F$›, (*simp add*: *hull.intros hull-closed*)+,
simp add: *gen-in*, *simp add*: *hull-closed*)
fix *p q w* **assume** $p \in \langle C \rangle$ $q \in \langle C \rangle$ $p \cdot w \in \langle C \rangle$ $w \cdot q \in \langle C \rangle$
have *eq*: $(Dec\ C\ p) \cdot (Dec\ C\ w \cdot q) = (Dec\ C\ p \cdot w) \cdot (Dec\ C\ q)$
using *code.code-dec-morph*[*OF* ‹*code C*› ‹$p \in \langle C \rangle$› ‹$w \cdot q \in \langle C \rangle$›, *unfolded*

lassoc]
unfolding *code.code-dec-morph*[*OF* ‹*code C*› ‹$p \cdot w \in \langle C\rangle$› ‹$q \in \langle C\rangle$›, *symmetric*].
have *Dec C p* $\bowtie$ *Dec C p* $\cdot$ *w*
using *eqd-comp*[*OF eq*].
hence *Dec C p* $\leq p$ *Dec C p* $\cdot$ *w*
using ‹$p \cdot w \in \langle C\rangle$› ‹$p \in \langle C\rangle$› *concat-morph decI prefD pref-antisym triv-pref*
unfolding *prefix-comparable-def*
by *metis*
then obtain *ts* **where** (*Dec C p*) $\cdot$ *ts* = *Dec C p* $\cdot$ *w*
using *lq-pref* **by** *blast*
hence *ts* $\in$ *lists C*
using *append-in-lists-conv*[*of Dec C p ts C*] *dec-dom'*[*OF* ‹$p \cdot w \in \langle C\rangle$›]
unfolding ‹(*Dec C p*) $\cdot$ *ts* = *Dec C p* $\cdot$ *w*› **by** *blast*
hence *concat ts* = *w*
using *concat-morph*[*of Dec C p ts*]
unfolding ‹(*Dec C p*) $\cdot$ *ts* = *Dec C p* $\cdot$ *w*› *decI*[*OF* ‹$p \cdot w \in \langle C\rangle$›] *decI*[*OF* ‹$p \in \langle C\rangle$›] **by** *auto*
thus $w \in \langle C\rangle$
using ‹*ts* $\in$ *lists C*› **by** *auto*
qed
qed
qed

That is, a code is its own free basis

lemma *code-free-basis*: **assumes** *code C* **shows** $C = \mathfrak{B}_F\ C$
using *basis-of-hull*[*of C*, *unfolded code-gen-free-hull*[*OF assms*, *symmetric*] *code.code-is-basis*[*OF assms*]]
unfolding *free-basis-def*.

Moreover, the free hull of G is the smallest code-generated hull containing G. In other words, the term free hull is appropriate.

First, several intuitive monotonicity and closure results.

lemma *free-hull-mono*: $G \subseteq H \Longrightarrow \langle G\rangle_F \subseteq \langle H\rangle_F$
proof
assume $G \subseteq H$
fix *x* **assume** $x \in \langle G\rangle_F$
have *el*: $\bigwedge w.\ w \in G \Longrightarrow w \in \langle H\rangle_F$
using ‹$G \subseteq H$› *free-hull.free-gen-in* **by** *auto*
show $x \in \langle H\rangle_F$
proof (*rule free-hull.induct*[*of x G*], *simp add*: ‹$x \in \langle G\rangle_F$›, *simp add*: *free-hull.intros*(*1*), *simp add*: *el*, *simp add*: *free-hull.intros*(*3*))
show $\bigwedge p\ q\ w.\ p \in \langle H\rangle_F \Longrightarrow q \in \langle H\rangle_F \Longrightarrow p \cdot w \in \langle H\rangle_F \Longrightarrow w \cdot q \in \langle H\rangle_F \Longrightarrow w \in \langle H\rangle_F$
using *free-hull.intros*(*4*) **by** *auto*
qed
qed

lemma *free-hull-idem*: $\langle\langle G\rangle_F\rangle_F = \langle G\rangle_F$
proof
  show $\langle\langle G\rangle_F\rangle_F \subseteq \langle G\rangle_F$
  proof
    fix x **assume** $x \in \langle\langle G\rangle_F\rangle_F$
    show $x \in \langle G\rangle_F$
    proof (*rule free-hull.induct*[*of* x $\langle G\rangle_F$], *simp add*: ‹$x \in \langle\langle G\rangle_F\rangle_F$›,
          simp add: *free-hull.intros*(*1*), *simp add*: *free-hull.intros*(*2*), *simp add*: *free-hull.intros*(*3*))
      show $\bigwedge p\ q\ w.\ p \in \langle G\rangle_F \Longrightarrow\ q \in \langle G\rangle_F \Longrightarrow\ p \cdot w \in \langle G\rangle_F \Longrightarrow\ w \cdot q \in \langle G\rangle_F \Longrightarrow w \in \langle G\rangle_F$
        using *free-hull.intros*(*4*) **by** *auto*
    qed
  qed
next
  show $\langle G\rangle_F \subseteq \langle\langle G\rangle_F\rangle_F$
    using *free-hull-hull hull-in-free-hull* **by** *auto*
qed

lemma *hull-gen-free-hull*: $\langle\langle G\rangle\rangle_F = \langle G\rangle_F$
proof
  show $\langle\langle G\rangle\rangle_F \subseteq \langle G\rangle_F$
    using *free-hull-idem free-hull-mono hull-in-free-hull* **by** *metis*
next
  show $\langle G\rangle_F \subseteq \langle\langle G\rangle\rangle_F$
    by (*simp add*: *free-hull-mono genset-sub*)
qed

Code is also the free basis of its hull.

lemma *code-free-basis-hull*: *code* $C \Longrightarrow C = \mathfrak{B}_F\ \langle C\rangle$
  unfolding *free-basis-def* **using** *code-free-basis*[*unfolded free-basis-def*]
  unfolding *hull-gen-free-hull*.

The minimality of the free hull easily follows.

theorem *free-hull-min*: **assumes** *code* C **and** $G \subseteq \langle C\rangle$ **shows** $\langle G\rangle_F \subseteq \langle C\rangle$
  using *free-hull-mono*[*OF* ‹$G \subseteq \langle C\rangle$›] **unfolding** *hull-gen-free-hull*
  unfolding *code-gen-free-hull*[*OF* ‹*code* C›].

theorem *free-hull-inter*: $\langle G\rangle_F = \bigcap\ \{M.\ G \subseteq M \wedge M = \langle M\rangle_F\}$
proof
  have $X \in \{M.\ G \subseteq M \wedge M = \langle M\rangle_F\} \Longrightarrow \langle G\rangle_F \subseteq X$ **for** X
    unfolding *mem-Collect-eq*[*of* - $\lambda\ M.\ G \subseteq M \wedge M = \langle M\rangle_F$]
    using *free-hull-mono*[*of* G X] **by** *simp*
  from *Inter-greatest*[*of* $\{M.\ G \subseteq M \wedge M = \langle M\rangle_F\}$, *OF this*]
  show $\langle G\rangle_F \subseteq \bigcap\ \{M.\ G \subseteq M \wedge M = \langle M\rangle_F\}$
    by *blast*
next
  show $\bigcap\ \{M.\ G \subseteq M \wedge M = \langle M\rangle_F\} \subseteq \langle G\rangle_F$
    by (*simp add*: *Inter-lower free-hull-idem genset-sub-free*)

qed

Decomposition into the free basis is a morphism.

lemma *free-basis-dec-morph*: $u \in \langle G \rangle_F \Longrightarrow v \in \langle G \rangle_F \Longrightarrow$
$Dec\ (\mathfrak{B}_F\ G)\ (u \cdot v) = (Dec\ (\mathfrak{B}_F\ G)\ u) \cdot (Dec\ (\mathfrak{B}_F\ G)\ v)$
using *code.code-dec-morph*[*OF free-basis-code*, *of u G v*, *symmetric*, *unfolded basis-gen-hull-free*[*of G*]].

3.7 Lists as the free hull of singletons

A crucial property of free monoids of words is that they can be seen as lists over the free basis, instead as lists over the original alphabet.

abbreviation *sings* **where** $sings\ B \equiv \{[b] \mid b.\ b \in B\}$

lemma *sings-image*: $sings\ B = (\lambda\ x.\ [x])\ `\ B$
using *Setcompr-eq-image*.

lemma *lists-sing-map-concat-ident*: $xs \in lists\ (sings\ B) \Longrightarrow xs = map\ (\lambda\ x.\ [x])\ (concat\ xs)$
by (*induct xs*, *simp*, *auto*)

lemma *code-sings*: *code* (*sings B*)
proof
fix *xs ys* **assume** *xs*: $xs \in lists\ (sings\ B)$ **and** *ys*: $ys \in lists\ (sings\ B)$
and *eq*: $concat\ xs = concat\ ys$
from *lists-sing-map-concat-ident*[*OF xs*, *unfolded eq*]
show $xs = ys$ **unfolding** *lists-sing-map-concat-ident*[*OF ys*, *symmetric*].
qed

lemma *sings-gen-lists*: $\langle sings\ B \rangle = lists\ B$
unfolding *hull-concat-lists*
proof(*intro equalityI subsetI*, *standard*)
fix *xs*
show $xs \in concat\ `\ lists\ (sings\ B) \Longrightarrow \forall\, x \in set\ xs.\ x \in B$
by *force*
assume $xs \in lists\ B$
hence $map\ (\lambda x.\ x\ \#\ \varepsilon)\ xs \in lists\ (sings\ B)$
by *force*
from *imageI*[*OF this*, *of concat*]
show $xs \in concat\ `\ lists\ (sings\ B)$
unfolding *concat-map-sing-ident*[*of xs*].
qed

lemma $sings\ B = \mathfrak{B}_F\ (lists\ B)$
using *code-free-basis-hull*[*OF code-sings*, *of B*, *unfolded sings-gen-lists*].

lemma *map-sings*: $xs \in lists\ B \Longrightarrow map\ (\lambda x.\ x\ \#\ \varepsilon)\ xs \in lists\ (sings\ B)$
by (*induct xs*) *auto*

lemma *dec-sings*: $xs \in lists\ B \Longrightarrow Dec\ (sings\ B)\ xs = map\ (\lambda\ x.\ [x])\ xs$
using *code.code-unique-dec*[*OF code-sings*, *of map* $(\lambda\ x.\ [x])$ *xs B*, *OF map-sings*]
unfolding *concat-map-sing-ident*.

end

theory *Periodicity-Lemma*
imports *CoWBasic*
begin

Chapter 4

The Periodicity Lemma

The Periodicity Lemma says that if a sufficiently long word has two periods p and q, then the period can be refined to *gcd p q*. The consequence is equivalent to the fact that the corresponding periodic roots commute. "Sufficiently long" here means at least $p + q - gcd\ p\ q$. It is also known as the Fine and Wilf theorem due to its authors [3].

If we relax the requirement to $p + q$, then the claim becomes easy, and it is proved in theory *Combinatorics-Words.CoWBasic* as *two-pers*: $[\![w \leq p\ u^\omega;\ w \leq p\ v^\omega;\ |u| + |v| \leq |w|]\!] \Longrightarrow u \cdot v = v \cdot u$.

theorem *per-lemma-relaxed*:
  assumes *periodN w p* **and** *periodN w q* **and** $p + q \leq |w|$
  shows $(take\ p\ w) \cdot (take\ q\ w) = (take\ q\ w) \cdot (take\ p\ w)$
  using *two-pers*[*OF*
    ‹*periodN w p*›[*unfolded periodN-def*[*of w p*]]
    ‹*periodN w q*›[*unfolded periodN-def*[*of w q*]], *unfolded*
    take-len[*OF add-leD1*[*OF* ‹$p + q \leq |w|$›]]
    take-len[*OF add-leD2*[*OF* ‹$p + q \leq |w|$›]], *OF* ‹$p + q \leq |w|$›].

Also in terms of the numeric period:

thm *two-periodsN*

4.1 Main claim

We first formulate the claim of the Periodicity lemma in terms of commutation of two periodic roots. For trivial reasons we can also drop the requirement that the roots are nonempty.

lemma *per-lemma-comm*:
  assumes $w \leq p\ r \cdot w$ **and** $w \leq p\ s \cdot w$
    and *len*: $|s| + |r| - (gcd\ |s|\ |r|) \leq |w|$
  shows $s \cdot r = r \cdot s$
  using *assms*

proof (*induction* $|s| + |s| + |r|$ *arbitrary*: w r s *rule*: *less-induct*)
case *less*
consider (*empty*) $s = \varepsilon$ | (*short*) $|r| < |s|$ | (*step*) $s \neq \varepsilon \wedge |s| \leq |r|$ **by** *force*
then show *?case*
proof (*cases*)
case (*empty*)
thus $s \cdot r = r \cdot s$ **by** *fastforce*
next
case (*short*)
thus $s \cdot r = r \cdot s$
using *less.hyps*[*OF* - ‹$w \leq_p s \cdot w$› ‹$w \leq_p r \cdot w$›
‹$|s| + |r| - (gcd\ |s|\ |r|) \leq |w|$›[*unfolded gcd.commute*[*of* $|s|$] *add.commute*[*of* $|s|$]]] **by** *fastforce*
next
case (*step*)
hence $s \neq \varepsilon$ **and** $|s| \leq |r|$ **by** *blast+*
from *le-add-diff*[*OF gcd-le2-nat*[*OF* ‹$s \neq \varepsilon$›[*folded length-0-conv*], *of* $|r|$], *unfolded gcd.commute*[*of* $|r|$]]
have $|r| \leq |w|$
using ‹$|s| + |r| - (gcd\ |s|\ |r|) \leq |w|$› *order.trans* **by** *fast*
hence $|s| \leq |w|$
using ‹$|s| \leq |r|$› *order.trans* **by** *blast*
from *pref-prod-long*[*OF* ‹$w \leq_p s \cdot w$› *this*]
have $s \leq_p w$**.**

obtain w' **where** $s \cdot w' = w$ **and** $|w'| < |w|$ **using** ‹$s \neq \varepsilon$› ‹$s \leq_p w$› **by** *auto*
have $w' \leq_p w$
using ‹$w \leq_p s \cdot w$› **unfolding** ‹$s \cdot w' = w$›[*symmetric*] *pref-cancel-conv*.
from *this*[*folded* ‹$s \cdot w' = w$›]
have $w' \leq_p s \cdot w'$**.**

have $s \leq_p r$
using *pref-prod-short*[*OF prefix-order.trans*[*OF* ‹$s \leq_p w$› ‹$w \leq_p r \cdot w$›] ‹$|s| \leq |r|$›].
hence $w' \leq_p (s^{-1>} r) \cdot w'$
using ‹$w \leq_p r \cdot w$› ‹$s \cdot w' = w$› *pref-prod-pref*[*OF* - ‹$w' \leq_p w$›, *of* $s^{-1>} r$] **by** *fastforce*

have *ind-len*: $|s| + |s^{-1>} r| - (gcd\ |s|\ |s^{-1>} r|) \leq |w'|$
using ‹$|s| + |r| - (gcd\ |s|\ |r|) \leq |w|$› ‹$s \cdot w' = w$› ‹$s \leq_p r$› **by** *auto*

have $s \cdot s^{-1>} r = s^{-1>} r \cdot s$
using *less.hyps*[*OF* - ‹$w' \leq_p (s^{-1>} r) \cdot w'$› ‹$w' \leq_p s \cdot w'$› *ind-len*] ‹$s \leq_p r$› ‹$|w'| < |w|$› **by** *force*

thus $s \cdot r = r \cdot s$
using ‹$s \leq_p r$› **by** *auto*
qed
qed

We can now prove the numeric version.

theorem *per-lemma*: **assumes** *periodN w p* **and** *periodN w q* **and** *len*: $p + q - gcd\ p\ q \leq |w|$
shows *periodN w* (*gcd p q*)
proof–
have *takep*: $w \leq p\ (take\ p\ w) \cdot w$ **and** *takeq*: $w \leq p\ (take\ q\ w) \cdot w$
using ‹*periodN w p*› ‹*periodN w q*› *periodN-D3* **by** *blast*+
have *lenp*: $|take\ p\ w| = p$
using *gcd-le2-nat*[*OF per-positive*[*OF* ‹*periodN w q*›], *of p*] *len take-len*
by *auto*
have *lenq*: $|take\ q\ w| = q$
using *gcd-le1-nat*[*OF per-positive*[*OF* ‹*periodN w p*›], *of q*] *len take-len*
by *simp*
obtain *t* **where** $take\ p\ w \in t*$ **and** $take\ q\ w \in t*$
using *comm-rootE*[*OF per-lemma-comm*[*OF takeq takep*, *unfolded lenp lenq*, *OF len*], *of thesis*] **by** *blast*
hence $w \leq p\ t^\omega$
using ‹*periodN w p*› *periodN-def per-root-trans* **by** *blast*
have *periodN w* $|t|$
using *root-periodN*[*OF per-nemp*[*OF* ‹*periodN w p*›] ‹$w \leq p\ t^\omega$›].
have $|t|$ *dvd* (*gcd p q*)
using *common-root-len-gcd*[*OF* ‹*take p w* ∈ *t**› ‹*take q w* ∈ *t**›] **unfolding** *lenp lenq*.
from *dvd-div-mult-self*[*OF this*]
have *gcd p q div* $|t| * |t| = gcd\ p\ q$.
have $gcd\ p\ q \neq 0$
using ‹*periodN w p*› **by** *auto*
from *this*[*folded dvd-div-eq-0-iff*[*OF* ‹$|t|$ *dvd* (*gcd p q*)›]]
show *periodN w* (*gcd p q*)
using *per-mult*[*OF* ‹*periodN w* $|t|$›, *of gcd p q div* $|t|$, *unfolded dvd-div-mult-self*[*OF* ‹$|t|$ *dvd* (*gcd p q*)›]] **by** *blast*
qed

4.2 Optimality

FW-word (where FW stands for Fine and Wilf) yields a word which show the optimality of the bound in the Periodicity lemma. Moreover, the obtained word has maximum possible letters (each equality of letters is forced by periods). The latter is not proved here.

term *butlast* ([*0..<*(*gcd p q*)]$^{@}$(*p div* (*gcd p q*)))·[*gcd p q*]·(*butlast* ([*0..<*(*gcd p q*)]$^{@}$(*p div* (*gcd p q*))))

— an auxiliary claim
lemma *ext-per-sum*: **assumes** *periodN w p* **and** *periodN w q* **and** $p \leq |w|$
shows *periodN* ((*take p w*) · *w*) (*p*+*q*)
proof–
have *nemp*: $take\ p\ w \cdot take\ q\ w \neq \varepsilon$

```
    using ‹periodN w p› by auto
  have take (p + q) (take p w · w) = take p (take p w · w) · take q (drop p (take
p w · w))
    using take-add by blast
  also have ... = take p w · take q w
    by (simp add: ‹p ≤ |w|›)
  ultimately have sum: take (p + q) (take p w · w) = take p w · take q w
    by simp
  show ?thesis
    using assms(1) assms(2) nemp
    unfolding periodN-def period-root-def sum rassoc same-prefix-prefix
    using pref-prolong  by blast
qed

abbreviation fw-p-per p q ≡ butlast ([0..<(gcd p q)]@(p div (gcd p q)))
abbreviation fw-base p q ≡ fw-p-per p q · [gcd p q]· fw-p-per p q

fun FW-word :: nat ⇒ nat ⇒ nat list where
  FW-word-def: FW-word p q =
— symmetry            (if q < p then  FW-word q p else
— artificial value    if p = 0 ∨ p = q then ε else
— base case            if gcd p q = q − p then fw-base p q
— step                else (take p (FW-word p (q−p))) · FW-word p (q−p)     )

lemma FW-sym: FW-word p q = FW-word q p
  by (cases rule: linorder-cases[of p q], simp+)

theorem fw-word': ¬ p dvd q ⟹ ¬ q dvd p ⟹
 |FW-word p q| = p + q − gcd p q − 1 ∧ periodN (FW-word p q) p ∧ periodN
(FW-word p q) q ∧ ¬ periodN (FW-word p q) (gcd p q)
proof (induction p + p + q arbitrary: p q rule: less-induct)
  case less
  have p ≠ 0
    using  ‹¬ q dvd p› dvd-0-right[of q] by meson
  have p ≠ q
    using ‹¬ p dvd q› by auto
  then consider q < p | p < q
    by linarith
  then show ?case
  proof (cases)
    assume q < p
    have less: q + q + p < p + p + q
      by (simp add: ‹q < p›)
    thus ?case
      using less.hyps[OF - ‹¬ q dvd p› ‹¬ p dvd q›]
     unfolding FW-sym[of p q] gcd.commute[of p q] add.commute[of p q] by blast
  next
    let ?d = gcd p q
    let ?dw = [0..<(gcd p q)]
```

let *?pd* = *p div* (*gcd p q*)
assume *p* < *q*
thus *?thesis*
proof (*cases ?d* = *q* − *p*)
assume *?d* = *q* − *p* **hence** *p* + *?d* = *q* **using** ‹*p* < *q*› **by** *auto*
have *fw*: *FW-word p q* = *fw-base p q*
using *FW-word-def* ‹*p* ≠ *0*› ‹*gcd p q* = *q* − *p*› ‹*p* < *q*› **by** *auto*

have *ppref*: |*butlast* (*?dw*[@] *?pd*)·[*?d*]| = *p*
using *length-append* ‹*p* ≠ *0*› *pow-len*[*of ?dw ?pd*]
by *auto*
note *ppref′* = *this*[*unfolded length-append*]
have *qpref*: |*butlast* (*?dw*[@] *?pd*)·[*?d*]·*?dw*| = *q*
unfolding *lassoc length-append ppref′* **using** ‹*p* + *gcd p q* = *q*› **by** *simp*
have *butlast* (*?dw*[@] *?pd*)·[*?d*] ≤*p FW-word p q*
unfolding *fw* **by** *force*
from *pref-take*[*OF this*]
have *takep*: *take p* (*FW-word p q*) = *butlast* (*?dw*[@] *?pd*)·[*?d*]
unfolding *ppref*.

have *?dw* ≠ *ε* **and** |*?dw*| = *?d*
using ‹*p* ≠ *0*› **by** *auto*
have *?pd* ≠ *0*
by (*simp add*: ‹*p* ≠ *0*› *dvd-div-eq-0-iff*)
from *not0-implies-Suc*[*OF this*]
obtain *e* **where** *?pd* = *Suc e* **by** *blast*
have *gcd p q* ≠ *p*
using ‹¬ *p dvd q*› *gcd-dvd2*[*of p q*] **by** *force*
hence *Suc e* ≠ *1*
using *dvd-mult-div-cancel*[*OF gcd-dvd1*[*of p q*], *unfolded* ‹*?pd* = *Suc e*›] **by** *force*
hence *e* ≠ *0* **by** *simp*

have [*0..*<*gcd p q*] [@] *e* ≠ *ε*
using ‹[*0..*<*gcd p q*] ≠ *ε*› ‹*e* ≠ *0*› *nonzero-pow-emp* **by** *blast*
hence *but-dec*: *butlast* (*?dw*[@] *?pd*) = *?dw* · *butlast* (*?dw*[@] *e*)
unfolding ‹*?pd* = *Suc e*› *pow-Suc-list butlast-append if-not-P*[*OF* ‹[*0..*<*gcd p q*] [@] *e* ≠ *ε*›] **by** *blast*
have *but-dec-b*: *butlast* (*?dw*[@] *?pd*) = *?dw*[@] *e* · *butlast ?dw*
unfolding ‹*?pd* = *Suc e*› *pow-Suc2-list butlast-append if-not-P*[*OF* ‹*?dw* ≠ *ε*›] **by** *blast*
have *butlast* (*?dw*[@] *?pd*)·[*?d*]·*?dw* ≤*p FW-word p q*
using *fw but-dec* **by** *auto*
note *takeq* = *pref-take*[*OF this*, *unfolded qpref*]

have |*FW-word p q*| = *p* + *q* − *gcd p q* − *1*
proof−
have |*?dw*| = *?d*
by *auto*

```
    have gcd p q dvd p
      by auto
    hence |?dw @ ?pd| = p
      using pow-len[of ?dw ?pd]
      by auto
    hence |butlast (?dw @ ?pd)| = p − 1
      by simp
    hence |FW-word p q| = (p − 1) + 1 + (p − 1)
      using fw unfolding length-append
      by auto
    thus |FW-word p q| = p + q − gcd p q − 1
      unfolding ‹gcd p q = q − p› using ‹p + gcd p q = q› ‹p ≠ 0› add.assoc
by auto
  qed

  have periodN (FW-word p q) p
  proof−
    have take p (FW-word p q) ≠ ε
      using ‹p ≠ 0› unfolding length-0-conv[symmetric] ppref[folded takep].
    thus periodN (FW-word p q) p
     unfolding periodN-def period-root-def takep unfolding fw lassoc by auto
  qed

  have periodN (FW-word p q) q
    unfolding periodN-def period-root-def
  proof
    show take q (FW-word p q) ≠ ε
      unfolding length-0-conv[symmetric] qpref[folded takeq] using ‹p ≠ 0› ‹p
< q› by linarith
  next
    show FW-word p q ≤p take q (FW-word p q) · FW-word p q
      unfolding takeq
        unfolding fw rassoc pref-cancel-conv but-dec but-dec-b ‹?pd = Suc e›
pow-Suc2-list butlast-append pow-Suc-list if-not-P[OF ‹?dw ≠ ε›]
   unfolding lassoc power-commutes[symmetric] unfolding rassoc pref-cancel-conv
      using pref-ext[OF prefixeq-butlast, of ?dw] by blast
  qed

  have ¬ periodN (FW-word p q) ?d
  proof−
    have last-a: last (take p (FW-word p q)) = ?d
      unfolding takep nth-append-length[of butlast (?dw @ ?pd) ?d ε]
        last-snoc by blast
    have ?dw ≤p FW-word p q
      using fw but-dec by simp
    from pref-take[OF this, unfolded ‹|?dw| = ?d›]
    have takegcd: take (gcd p q) (FW-word p q) = [0..<gcd p q].
     have fw-dec-b: FW-word p q = [0..<gcd p q]@e · butlast ([0..<gcd p q]) ·
[?d] · (butlast ([0..<gcd p q]@(p div gcd p q)))
```

```
      using fw but-dec-b by simp
    hence gcdepref: [0..<gcd p q]@ Suc e ≤p take (gcd p q) (FW-word p q) ·
FW-word p q
      using takegcd by simp
    have |[0..<gcd p q]@ Suc e| = p
      unfolding pow-len ‹|?dw| = ?d› ‹?pd = Suc e›[symmetric] using
        dvd-div-mult-self[OF gcd-dvd1].
    from pref-take[OF gcdepref, unfolded this]
    have takegcdp: take p (take (gcd p q) (FW-word p q) · (FW-word p q)) =
[0..<gcd p q]@e · [0..<gcd p q]
      unfolding pow-Suc2-list.
    have 0 < gcd p q by (simp add: ‹p ≠ 0›)
    from last-upt[OF this]
    have last-b: last (take p (take (gcd p q) (FW-word p q) · (FW-word p q)))
= gcd p q − 1
      unfolding takegcdp last-appendR[of [0..<gcd p q] [0..<gcd p q]@e, OF
‹[0..<gcd p q] ≠ ε›].
    have p ≤ |FW-word p q|
      using ‹|FW-word p q| = p + q − gcd p q − 1› ‹gcd p q = q − p› ‹p < q›
by linarith
    have gcd p q ≠ gcd p q − 1
      using ‹gcd p q = q − p› ‹p < q› by linarith
    hence take p (FW-word p q) ≠ take p (take (gcd p q) (FW-word p q) ·
(FW-word p q))
    unfolding last-b[symmetric] unfolding last-a[symmetric] using arg-cong[of
- - last] by blast
    hence ¬ FW-word p q ≤p take (gcd p q) (FW-word p q) · FW-word p q
    using pref-share-take[OF - ‹p ≤ |FW-word p q|›, of take (gcd p q) (FW-word
p q) · FW-word p q] by blast
    thus ¬ periodN (FW-word p q) (gcd p q)
      unfolding periodN-def period-root-def by blast
  qed

  show ?thesis
    using ‹periodN (FW-word p q) p› ‹periodN (FW-word p q) q›
      ‹|FW-word p q| = p + q − gcd p q − 1› ‹¬ periodN (FW-word p q) (gcd
p q)› by blast
 next
  let ?d′ = gcd p (q−p)
  assume gcd p q ≠ q − p
  hence fw: FW-word p q = (take p (FW-word p (q−p))) · FW-word p (q−p)
      using FW-word-def ‹p ≠ 0› ‹p ≠ q› ‹p < q› by (meson less-Suc-eq
not-less-eq)

  have divhyp1: ¬ p dvd q − p
    using ‹¬ p dvd q› ‹p < q› dvd-minus-self by auto

  have divhyp2: ¬ q − p dvd p
  proof (rule notI)
```

```
      assume q − p dvd p
      have q = p + (q − p)
        by (simp add: ‹p < q› less-or-eq-imp-le)
      from gcd-add2[of p q − p, folded this, unfolded gcd-nat.absorb2[of q − p p,
OF ‹q − p dvd p›]]
      show False
        using ‹gcd p q ≠ q − p› by blast
    qed

    have lenhyp: p + p + (q − p) < p + p + q
      using ‹p < q› ‹p ≠ 0› by linarith

— induction assumption
    have |FW-word p (q − p)| = p + (q − p) − ?d′ − 1 and periodN (FW-word
p (q−p)) p and periodN (FW-word p (q−p)) (q−p) and
      ¬ periodN (FW-word p (q−p)) (gcd p (q−p))
      using less.hyps[OF - divhyp1 divhyp2] lenhyp
      by blast+

— auxiliary facts
    have p + (q − p) = q
      using divhyp1 dvd-minus-self by auto
    have ?d = ?d′
     using gcd-add2[of p q−p, unfolded le-add-diff-inverse[OF less-imp-le[OF ‹p
< q›]]].
    have ?d ≠ q
      using ‹¬ q dvd p› gcd-dvd2[of q p, unfolded gcd.commute[of q]] by force
    from this[unfolded nat-neq-iff]
    have ?d < q
      using gr-implies-not0 ‹p < q› nat-dvd-not-less by blast
    hence 1 ≤ q − ?d
      by linarith
    have ?d′ < q − p
      using gcd-le2-nat[OF per-positive[OF ‹periodN (FW-word p (q − p)) (q −
p)›], of p] divhyp2[unfolded gcd-nat.absorb-iff2] nat-less-le by blast
    hence p ≤ |(FW-word p (q − p))|
      unfolding ‹|FW-word p (q − p)| = p + (q − p) − ?d′ − 1› diff-diff-left
discrete by linarith
    have FW-word p (q − p) ≠ ε
      unfolding length-0-conv[symmetric] using ‹p ≤ |FW-word p (q − p)|› ‹p
≠ 0›[folded le-zero-eq]
      by linarith

— claim 1
    have |FW-word p q| = p + q − ?d − 1
    proof−
      have |FW-word p q| = |take p (FW-word p (q − p))| + |FW-word p (q −
p)|
        using fw length-append[of take p (FW-word p (q − p)) FW-word p (q −
```

```
p)]
        by presburger
      also have ... = p + (p + (q − p) − ?d' − 1)
        unfolding ‹|FW-word p (q − p)| = p + (q − p) − ?d' − 1›
          take-len[OF ‹p ≤ |FW-word p (q − p)|›] by blast
      also have ... = p + (q − ?d  − 1)
        unfolding ‹?d = ?d'› using  ‹p < q› by auto
      also have ... = p + (q − ?d) − 1
        using Nat.add-diff-assoc[OF ‹1 ≤ q − ?d›].
      also have ... = p + q − ?d − 1
        by (simp add: ‹?d < q› less-or-eq-imp-le)
      finally show |FW-word p q| = p + q − ?d − 1
        by presburger
    qed

— claim 2
    have periodN (FW-word p q) p
      using fw  ext-per-left[OF ‹periodN (FW-word p (q−p)) p› ‹p ≤ |FW-word
p (q − p)|›]
      by presburger

— claim 3
    have periodN (FW-word p q) q
     using ext-per-sum[OF ‹periodN (FW-word p (q − p)) p› ‹periodN (FW-word
p (q − p)) (q − p)› ‹p ≤ |FW-word p (q − p)|›, folded fw, unfolded ‹p + (q−p) =
q›].

— claim 4
    have ¬ periodN (FW-word p q) ?d
      using ‹¬ periodN (FW-word p (q −p)) (gcd p (q−p))›
      unfolding ‹?d = ?d'›[symmetric]
      using periodN-fac[of take p (FW-word p (q − p)) FW-word p (q − p) ε ?d,
unfolded append-Nil2,
                        OF - ‹FW-word p (q − p) ≠ ε›, folded fw] by blast
    thus ?thesis
      using ‹periodN (FW-word p q) p› ‹periodN (FW-word p q) q› ‹|FW-word p
q| = p + q − ?d − 1› by blast
  qed
 qed
qed

theorem fw-word: assumes ¬ p dvd q ¬ q dvd p
 shows |FW-word p q| = p + q − gcd p q − 1 and periodN (FW-word p q) p
and  periodN (FW-word p q) q and ¬ periodN (FW-word p q) (gcd p q)
 using fw-word'[OF assms] by blast+
```

Calculation examples

```
value FW-word 3 7
value FW-word 5 7
```

value *FW-word 5 13*
value *FW-word 4 6*
value *FW-word 12 18*

4.3 Other variants of the periodicity lemma

Periodicity lemma is one of the most frequent tools in Combinatorics on words. Here are some useful variants.

lemma *fac-two-conjug-prim-root*:
assumes *facs*: $u \leq_f r^{@}k$ $u \leq_f s^{@}l$ **and** *nemps*: $r \neq \varepsilon$ $s \neq \varepsilon$ **and** *len*: $|r| + |s| - gcd\ (|r|)\ (|s|) \leq |u|$
shows $\varrho\ r \sim \varrho\ s$
proof −
obtain r' s' **where** *prefr'*: $u \leq_p r'^{@}k$ **and** *prefs'*: $u \leq_p s'^{@}l$
and *conjugs*: $r \sim r'$ $s \sim s'$
using *facs* **by** (*elim fac-pow-pref-conjug*)
have *rootr'*: $u \leq_p r' \cdot u$ **and** *roots'*: $u \leq_p s' \cdot u$
using *pref-prod-root*[*OF prefr'*] *pref-prod-root*[*OF prefs'*].
have *nemps'*: $r' \neq \varepsilon$ $s' \neq \varepsilon$ **using** *nemps conjugs conjug-nemp-iff* **by** *blast*+
have $|r'| + |s'| - gcd\ (|r'|)\ (|s'|) \leq |u|$ **using** *len*
unfolding *conjug-len*[*OF* ‹$r \sim r'$›] *conjug-len*[*OF* ‹$s \sim s'$›].
from *per-lemma-comm*[*OF roots' rootr' this*] **have** $r' \cdot s' = s' \cdot r'$.
then have $\varrho\ r' = \varrho\ s'$ **using** *comm-primroots*[*OF nemps'*] **by** *force*
also have $\varrho\ s \sim \varrho\ s'$ **using** *conjug-prim-root*[*OF* ‹$s \sim s'$› ‹$s \neq \varepsilon$›].
also have [*symmetric*]: $\varrho\ r \sim \varrho\ r'$ **using** *conjug-prim-root*[*OF* ‹$r \sim r'$› ‹$r \neq \varepsilon$›].
finally show $\varrho\ r \sim \varrho\ s$..
qed

lemma *fac-two-conjug-prim-root'*:
assumes *facs*: $u \leq_f r^{@}k$ $u \leq_f s^{@}l$ **and** $u \neq \varepsilon$ **and** *len*: $|r| + |s| - gcd\ (|r|)\ (|s|) \leq |u|$
shows $\varrho\ r \sim \varrho\ s$
proof −
have *nemps*: $r \neq \varepsilon$ $s \neq \varepsilon$ **using** *facs* ‹$u \neq \varepsilon$› **by** *auto*
show *conjugate* $(\varrho\ r)\ (\varrho\ s)$ **using** *fac-two-conjug-prim-root*[*OF facs nemps len*].
qed

lemma *fac-two-nconj-prim-pow*:
assumes *prims*: *primitive r primitive s* **and** $\neg\ r \sim s$
and *facs*: $u \leq_f r^{@}k$ $u \leq_f s^{@}l$
shows $|u| < |r| + |s| - gcd\ (|r|)\ (|s|)$
using ‹$\neg\ r \sim s$› *fac-two-conjug-prim-root*[*OF facs prim-nemp prim-nemp leI*, *OF prims*]
unfolding *prim-self-root*[*OF* ‹*primitive r*›] *prim-self-root*[*OF* ‹*primitive s*›]
by (*rule contrapos-np*)

end

theory *Lyndon-Schutzenberger*
  imports *CoWBasic*
begin

Chapter 5

Lyndon-Schützenberger Equation

The Lyndon-Schützenberger equation is the following equation on words:

$$x^a y^b = z^c,$$

in this formalization denoted as $x \,^{@}\, a \cdot y \,^{@}\, b = z \,^{@}\, c$, with $2 \leq a, b, c$. We formalize here a proof that the equation has periodic solutions only in free monoids, that is, that any three words x, y and z satisfying the equality pairwise commute. The result was first proved in [7] in a more general setting of free groups. There are several proofs to be found in the literature (for instance [4, 2]). The presented proof is the author's proof.

We set up a locale representing the Lyndon-Schützenberger Equation.

locale *LS* =
  fixes *x a y b z c*
  assumes *a*: $2 \leq a$ **and** *b*: $2 \leq b$ **and** *c*: $2 \leq c$ **and** *eq*: $x^{@}a \cdot y^{@}b = z^{@}c$
begin

lemma *a0*: $a \neq 0$ **and** *b0*: $b \neq 0$
  using *a b* **by** *auto*

If x^a or y^b is sufficiently long, then the calim follows from the Periodicity Lemma.

lemma *per-lemma-case*:
  assumes $|z| + |x| \leq |x^{@}a|$
  shows $x{\cdot}y{=}y{\cdot}x$
proof (*cases* $x = \varepsilon$)
  assume $x = \varepsilon$
  thus $x{\cdot}y{=}y{\cdot}x$ **by** *simp*
next
  assume $x \neq \varepsilon$
  hence $z^{@}c \neq \varepsilon$

using *eq assms emp-pow*[*of c*] **by** *auto*
hence $x^{@}a \leq p\ (z^{@}c)^{\omega}$
unfolding *period-root-def* **using**
pref-ext[*OF triv-pref*[*of* $x^{@}a\ y^{@}b$, *unfolded eq*], *of* $x^{@}a$] **by** *blast*
have $x\ ^{@}\ a \leq p\ x^{\omega}$
using ‹$x \neq \varepsilon$› *a0 root-self*[*THEN per-drop-exp*] **by** *blast*
from *two-pers*[*OF per-drop-exp*[*OF* ‹$x^{@}a \leq p\ (z^{@}c)^{\omega}$›] *this assms*]
have $z \cdot x = x \cdot z$**.**
hence $z^{@}c{\cdot}x^{@}a = x^{@}a{\cdot}z^{@}c$
by (*simp add*: *comm-add-exps*)
from *this*[*folded eq*, *unfolded rassoc cancel*, *symmetric*]
have $x^{@}a \cdot y^{@}b = y^{@}b \cdot x^{@}a$**.**
from *this*[*unfolded comm-pow-roots*[*OF a0 b0*]]
show $x \cdot y = y \cdot x$**.**
qed

The most challenging case is when $c = 3$.

lemma *core-case*:
assumes
$c = 3$ **and**
$b{*}|y| \leq a{*}|x|$ **and** $x \neq \varepsilon$ **and** $y \neq \varepsilon$ **and**
lenx: $a{*}|x| < |z| + |x|$ **and**
leny: $b{*}|y| < |z| + |y|$
shows $x{\cdot}y = y{\cdot}x$
proof–
have $a \neq 0$ **and** $b \neq 0$
using *a b* **by** *auto*

— We first show that a = 2
have $a{*}|x|{+}b{*}|y| = 3{*}|z|$
using ‹$c = 3$› *eq length-append*[*of* $x^{@}a\ y^{@}b$]
by (*simp add*: *pow-len*)
hence $3{*}|z| \leq a{*}|x| + a{*}|x|$
using ‹$b{*}|y| \leq a{*}|x|$› **by** *simp*
hence $3{*}|z| < 2{*}|z| + 2{*}|x|$
using *lenx* **by** *linarith*
hence $|z| + |x| < 3 * |x|$ **by** *simp*
from *less-trans*[*OF lenx this*, *unfolded mult-less-cancel2*]
have $a = 2$ **using** *a* **by** *force*

hence $|y| \leq |x|$ **using** ‹$b{*}|y| \leq a{*}|x|$› *b*
pow-len[*of x 2*] *pow-len*[*of y b*] *mult-le-less-imp-less*[*of a b* $|x|$ $|y|$] *not-le*
by *auto*
have $x{\cdot}x{\cdot}y^{@}b = z{\cdot}z{\cdot}z$ **using** *a eq* ‹*c*=*3*› ‹*a*=*2*›
by (*simp add*: *numeral-2-eq-2 numeral-3-eq-3*)

— Find words u, v, w
have $|z| < |x{\cdot}x|$
using ‹$|z| + |x| < 3 * |x|$› *add.commute* **by** *auto*

from *ruler-le*[*THEN prefD*, *OF triv-pref*[*of z z·z*] - *less-imp-le*[*OF this*]]
obtain *w* **where** *z·w = x·x*
using *prefI*[*of x·x y*$^{@}$*b z·z·z*, *unfolded rassoc*, *OF ‹x·x·y*$^{@}$*b = z·z·z›*] **by** *fastforce*

have $|x| < |z|$
using *‹a = 2›* *lenx* **by** *auto*
from *ruler-le*[*THEN prefD*, *OF* - - *less-imp-le*[*OF this*], *of x·x·y*$^{@}$*b*, *OF triv-pref*, *unfolded ‹x·x·y*$^{@}$*b = z·z·z›*, *OF triv-pref*]
obtain *u* :: *'a list* **where** *x·u=z*
by *blast*
have $u \neq \varepsilon$
using *‹|x| < |z|›* *‹x·u = z›* **by** *auto*
have *x = u·w* **using** *‹z·w = x·x›* *‹x·u = z›* **by** *auto*

have *|x·x| < |z·z|* **by** (*simp add*: *‹|x| < |z|›* *add-less-mono*)
from *ruler-le*[*OF triv-pref*[*of x·x y*$^{@}$*b*, *unfolded rassoc ‹x·x·y*$^{@}$*b = z·z·z›*, *unfolded lassoc*] *triv-pref*, *OF less-imp-le*[*OF this*]]
have *z·w* $\leq p$ *z·z*
unfolding *‹z·w = x·x›*.
obtain *v* :: *'a list* **where** *w · v = x*
using *lq-pref*[*of w x*]
pref-prod-pref'[*OF pref-cancel*[*OF ‹z·w* $\leq p$ *z·z›*, *folded ‹x · u = z›*, *unfolded ‹x = u · w› rassoc*], *folded ‹x = u · w›*] **by** *blast*
have *u·w·v* $\neq \varepsilon$
by (*simp add*: *‹u* $\neq \varepsilon$*›*)

— Express x, y and z in terms of u, v and w
hence *z = w·v·u*
using *‹w · v = x› ‹x · u = z›* **by** *auto*
from *‹x · x · y*$^{@}$*b = z · z · z›*[*unfolded this lassoc*, *folded ‹z · w = x · x›*, *unfolded this rassoc*]
have *w·v · u·w · y*$^{@}$*b = w·v·u·w·v·u·w·v·u*.
hence *y*$^{@}$*b = v·u·w·v·u*
using *pref-cancel* **by** *auto*

— Double period of uwv
from *periodN-fac*[*OF* - *‹u·w·v* $\neq \varepsilon$*›*, *of v u |y|*, *unfolded rassoc*, *folded this*]
have *periodN* (*u·w·v*) *|y|*
using *pow-per*[*OF ‹y* $\neq \varepsilon$*›* *b0*] **by** *blast*
have *u·w·v = x ·v*
by (*simp add*: *‹x = u · w›*)
have *u·w·v = u· x*
by (*simp add*: *‹w · v = x›*)
have *u·w·v* $\leq p$ u^{ω}
unfolding *period-root-def*
using *‹u · w · v = u · x›*[*unfolded ‹x = u · w›*] *‹u* $\neq \varepsilon$*›* *triv-pref*[*of u · u · w v*]
by *force*
have *periodN* (*u·w·v*) *|u|*
using *‹u · w · v* $\leq p$ *u* $^{\omega}$*›* **by** *auto*

— Common period d
obtain d::*nat* **where** d=*gcd* $|y|$ $|u|$
by *simp*
have $|y| + |u| \leq |u{\cdot}w{\cdot}v|$ **using** ‹$|y| \leq |x|$› *length-append* ‹$u{\cdot}w{\cdot}v = u{\cdot}\ x$›
by *simp*
hence *periodN* $(u{\cdot}w{\cdot}v)$ d
using ‹*periodN* $(u \cdot w \cdot v)$ $|u|$› ‹*periodN* $(u \cdot w \cdot v)$ $|y|$› ‹d = *gcd* $|y|$ $|u|$› *two-periodsN*
by *blast*

— Divisibility
have $v{\cdot}u{\cdot}z{=}y^{@}b$
by (*simp add*: ‹$y^{@}b = v \cdot u \cdot w \cdot v \cdot u$› ‹$z = w \cdot v \cdot u$›)
have $|u| = |v|$
using ‹$x = u \cdot w$› ‹$w \cdot v = x$› *length-append*[*of u w*] *length-append*[*of w v*] *add.commute*[*of* $|u|$ $|w|$] *add-left-cancel*
by *simp*
hence d *dvd* $|v|$ **using** *gcd-nat.cobounded1*[*of* $|v|$ $|y|$] *gcd.commute*[*of* $|y|$ $|u|$]
by (*simp add*: ‹d = *gcd* $|y|$ $|u|$›)
have d *dvd* $|u|$
by (*simp add*: ‹d = *gcd* $|y|$ $|u|$›)
have $|z| + |u| + |v| = b{*}|y|$
using *lenarg*[*OF* ‹$v{\cdot}u{\cdot}z{=}y^{@}b$›, *unfolded length-append pow-len*] **by** *auto*
from *dvd-add-left-iff*[*OF* ‹d *dvd* $|v|$›, *of* $|z|{+}|u|$, *unfolded this dvd-add-left-iff*[*OF* ‹d *dvd* $|u|$›, *of* $|z|$]]
have d *dvd* $|z|$
using ‹d = *gcd* $|y|$ $|u|$› *dvd-mult* **by** *blast*
from *lenarg*[*OF* ‹$z = w \cdot v \cdot u$›, *unfolded length-append pow-len*]
have d *dvd* $|w|$
using ‹d *dvd* $|z|$› ‹d *dvd* $|u|$› ‹d *dvd* $|v|$› **by** (*simp add*: *dvd-add-left-iff*)
hence d *dvd* $|x|$
using ‹d *dvd* $|v|$› ‹$w \cdot v = x$› **by** *force*

— x and y commute
have $x \leq p\ u{\cdot}w{\cdot}v$
by (*simp add*: ‹$x = u \cdot w$›)
have *periodN x d* **using** *per-pref'*[*OF* ‹$x{\neq}\varepsilon$› ‹*periodN* $(u{\cdot}w{\cdot}v)$ d› ‹$x \leq p\ u \cdot w{\cdot}v$›].
hence $x \in (take\ d\ x){*}$
using ‹d *dvd* $|x|$›
using *root-divisor* **by** *blast*

hence *periodN u d* **using** ‹$x = u \cdot w$› *per-pref'*
using ‹*periodN x d*› ‹$u \neq \varepsilon$› **by** *blast*
have *take d x* = *take d u* **using** ‹$u{\neq}\varepsilon$› ‹$x = u \cdot w$› *pref-share-take*
by (*simp add*: ‹d = *gcd* $|y|$ $|u|$›)
from *root-divisor*[*OF* ‹*periodN u d*› ‹d *dvd* $|u|$›, *folded this*]
have $u \in (take\ d\ x){*}$.

hence $z \in (take\ d\ x)*$
using ‹$x \cdot u = z$› ‹$x \in (take\ d\ x)*$› *add-roots* **by** *blast*
from *root-pref-cancel'*[*OF - root-pow-root*[*OF* ‹$x \in take\ d\ x*$›, *of a*],*of* $y^{@}b$, *unfolded eq*, *OF root-pow-root*[*OF this*, *of c*]]
have $y^{@}b \in (take\ d\ x)*$.
from *commD*[*OF root-pow-root*[*OF* ‹$x \in take\ d\ x*$›, *of a*] *this*]
show $x \cdot y = y \cdot x$
unfolding *comm-pow-roots*[*OF* ‹$a \neq 0$› ‹$b \neq 0$›, *of x y*].
qed

end — locale LS

The main proof is by induction on the length of z. It also uses the reverse symmetry of the equation which is exploited by two interpretations of the locale *LS*. Note also that the case $|x^a| < |y^b|$ is solved by using induction on $|z| + |y^b|$ instead of just on $|z|$.

lemma *Lyndon-Schutzenberger'*:
⟦ $x^{@}a \cdot y^{@}b = z^{@}c$; $2 \leq a$; $2 \leq b$; $2 \leq c$ ⟧
⟹ $x \cdot y = y \cdot x$
proof (*induction* $|z| + b* |y|$ *arbitrary*: *x y z a b c* *rule*: *less-induct*)
case *less*

interpret *LS x a y b z c* **using** ‹ $x^{@}a \cdot y^{@}b = z^{@}c$› ‹ $2 \leq a$ › ‹ $2 \leq b$› ‹ $2 \leq c$›
by (*simp add*: *LS.intro*)
interpret *LSrev*: *LS rev y b rev x a rev z c*
using *LS.intro*[*OF b a c*, *of rev y rev x rev z*, *folded rev-append rev-pow*, *unfolded rev-is-rev-conv*, *OF* ‹$x^{@}a \cdot y^{@}b = z^{@}c$›].

have *leneq*: $a * |x| + b*|y| = c * |z|$
using *eq* **unfolding** *pow-len*[*symmetric*] *length-append*[*symmetric*] **by** *simp*
have $a \neq 0$ **and** $b \neq 0$
using *a b* **by** *auto*

show $x \cdot y = y \cdot x$
proof (*cases* $x = \varepsilon \vee y = \varepsilon$)
show $x = \varepsilon \vee y = \varepsilon \Longrightarrow x \cdot y = y \cdot x$
by *auto*
next
assume $\neg\ (x = \varepsilon \vee y = \varepsilon)$ **hence** $x \neq \varepsilon$ **and** $y \neq \varepsilon$ **by** *blast+*
show $x \cdot y = y \cdot x$
proof (*cases* $|x\ @\ a| < |y\ @\ b|$)

— WLOG assumption
assume $|x^{@}a| < |y^{@}b|$
have $|rev\ z| + a* |rev\ x| < |z| + b* |y|$ **using** ‹$|x^{@}a| < |y^{@}b|$›
by (*simp add*: *pow-len*)
from *less.hyps*[*OF this LSrev.eq b a c*, *symmetric*]
show $x \cdot y = y \cdot x$

unfolding *rev-append*[*symmetric*] *rev-is-rev-conv* **by** *simp*
next
assume $\neg\ |x^{@}a| < |y^{@}b|$ **hence** $|y^{@}b| \leq |x^{@}a|$ **by** *force*
— case solved by the Periodicity lemma
consider (*with-Periodicity-lemma*)
$|z| + |x| \leq |x\ ^{@}\ a| \vee |z| + |y| \leq |y\ ^{@}\ b|$ |
(*without-Periodicity-lemma*)
$|x^{@}a| < |z| + |x|$ **and** $|y^{@}b| < |z| + |y|$
using *not-le-imp-less* **by** *blast*
thus $x \cdot y = y \cdot x$
proof (*cases*)
case *with-Periodicity-lemma*
assume *short*: $|z| + |x| \leq |x^{@}a| \vee |z| + |y| \leq |y^{@}b|$
have $x = \varepsilon \vee rev\ y = \varepsilon \Longrightarrow x \cdot y = y \cdot x$
by *auto*
thus $x \cdot y = y \cdot x$
using *per-lemma-case LSrev.per-lemma-case short*
unfolding *length-rev rev-append*[*symmetric*] *rev-is-rev-conv rev-pow*[*symmetric*]
by *blast*
next
case *without-Periodicity-lemma*
assume *lenx*: $|x^{@}a| < |z| + |x|$ **and** *leny*: $|y^{@}b| < |z| + |y|$
have *ex*: $\exists\ k.\ d = Suc\ (Suc\ k)$ **if** $2 \leq d$ **for** d
using *nat-le-iff-add that* **by** *auto*
have $a * |x| + b*|y| < 4 * |z|$
using *ex*[*OF* ‹$2 \leq a$›] *ex*[*OF* ‹$2 \leq b$›] *lenx leny* **unfolding** *pow-len* **by**
auto
hence $c < 4$ **using** *leneq* **by** *auto*
consider (*c-is-3*) $c = 3$ | (*c-is-2*) $c = 2$
using ‹$c < 4$› c **by** *linarith*
then show $x \cdot y = y \cdot x$
proof(*cases*)
case *c-is-3*
show $x \cdot y = y \cdot x$
using
core-case[*OF* ‹$c = 3$› ‹$|y^{@}b| \leq |x^{@}a|$›[*unfolded pow-len*] - - *lenx*[*unfolded*
pow-len] *leny*[*unfolded pow-len*]]
‹$x \neq \varepsilon$› ‹$y \neq \varepsilon$›
by *blast*
next
assume $c = 2$
hence *eq2*: $x^{@}a \cdot y^{@}b = z \cdot z$
by (*simp add*: *eq pow2-list*)
from *dual-order.trans le-cases*[*of* $|x^{@}a|$ $|z|$ $|z| \leq |x^{@}a|$, *unfolded*
eq-len-iff[*OF this*]]
have $|z| \leq |x^{@}a|$
using ‹$|y^{@}b| \leq |x^{@}a|$› **by** *blast*
obtain a' **where** $Suc\ a' = a$ **and** $1 \leq a'$
using ‹$2 \leq a$› *ex* **by** *auto*

from *eq2*[*folded* ‹*Suc a′ = a*›, *unfolded pow-Suc2-list rassoc*] *pow-Suc2-list*[*of x a′*, *unfolded this*, *symmetric*]
have *eq3*: $x \,^{@}\, a' \cdot x \cdot y \,^{@}\, b = z \cdot z$ **and** *aa′*:$x \,^{@}\, a' \cdot x = x \,^{@}\, a$ **.**
hence $|x^{@}a'| < |z|$
using ‹*Suc a′ = a*› *lenx pow-len* **by** *auto*
hence $|x| < |z|$
using *mult-le-mono*[*of 1 a′* $|z|$ $|x|$, *OF* ‹*1 ≤ a′*›, *THEN leD*] **unfolding** *pow-len*
by *linarith*
obtain *u w* **where** $x^{@}a'{\cdot}u = z$ **and** $w \cdot y^{@}b = z$
using *eqd-prefE*[*OF eq3*[*unfolded rassoc*] *less-imp-le*[*OF* ‹$|x^{@}a'| < |z|$›], *of thesis*]
eqd-prefE[*OF eq2*[*symmetric*] ‹$|z| \le |x^{@}a|$›, *of thesis*] **by** *fast*

have $x^{@}a'{\cdot}x{\cdot}y^{@}b = x^{@}a'{\cdot}u{\cdot}w{\cdot}y^{@}b$
unfolding *lassoc* ‹$x \,^{@}\, a' \cdot u = z$› ‹$w \cdot y^{@}b = z$› *aa′ cancel eq2* **by** *simp*
hence *u·w=x*
by *auto*
hence $|w{\cdot}u| = |x|$
using *swap-len* **by** *blast*

— Induction step: new equation with shorter z
have $w^{@}2{\cdot}y^{@}b = (w{\cdot}u)^{@}a$
unfolding *pow2-list* **using** ‹$w \cdot y \,^{@}\, b = z$› ‹$x \,^{@}\, a' \cdot u = z$› ‹*u·w=x*› *pow-slide*[*of w u a′*, *unfolded* ‹*Suc a′ = a*›] **by** *simp*
from *less.hyps*[*OF - this - b a*, *unfolded* ‹$|w{\cdot}u| = |x|$›]
have $y{\cdot}w = w{\cdot}y$
using ‹$|x| < |z|$› **by** *force*

have $y \cdot z = z \cdot y$
unfolding ‹$w \cdot y^{@}b = z$›[*symmetric*] *lassoc* ‹*y·w = w·y*›
by (*simp add*: *pow-commutes-list*)
hence $z^{@}c{\cdot}y^{@}b = y^{@}b{\cdot}z^{@}c$
by (*simp add*: *comm-add-exps*)
from *this*[*folded eq*, *unfolded lassoc*]
have $x^{@}a{\cdot}y^{@}b = y^{@}b{\cdot}x^{@}a$
using *cancel-right* **by** *blast*
from *this*[*unfolded comm-pow-roots*[*OF* ‹*a≠0*› ‹$b \neq 0$›]]
show $x \cdot y = y \cdot x$**.**
qed
qed
qed
qed
qed

theorem *Lyndon-Schutzenberger*:
assumes $x^{@}a{\cdot}y^{@}b = z^{@}c$ **and** $2 \le a$ **and** $2 \le b$ **and** $2 \le c$
shows *x·y = y·x* **and** *x·z = z·x* **and** *y·z = z·y*
proof–

show $x \cdot y = y \cdot x$
using *Lyndon-Schutzenberger'*[*OF assms*]**.**
have $c \neq 0$ **and** $b \neq 0$ **using** ‹$2 \leq c$› ‹$2 \leq b$› **by** *auto*
have $x \cdot x^{@}a \cdot y^{@}b = x^{@}a \cdot y^{@}b \cdot x$ **and** $y \cdot x^{@}a \cdot y^{@}b = x^{@}a \cdot y^{@}b \cdot y$
unfolding *comm-add-exp*[*OF* ‹$x \cdot y = y \cdot x$›[*symmetric*], *of b*] *comm-add-exp*[*OF* ‹$x \cdot y = y \cdot x$›, *symmetric*, *of a*]
lassoc power-commutes **by** *blast+*
thus $x{\cdot}z = z{\cdot}x$ **and** $y{\cdot}z = z{\cdot}y$
using *comm-drop-exp*[*OF* ‹$c \neq 0$›] **unfolding** *lassoc* ‹$x^{@}a{\cdot}y^{@}b = z^{@}c$› **by** *force+*
qed

end

References

[1] C. Choffrut and J. Karhumäki. *Combinatorics of Words*, pages 329–438. Springer-Verlag, Berlin, Heidelberg, 1997.

[2] P. Dömösi and G. Horváth. Alternative proof of the Lyndon–Schützenberger theorem. *Theoret. Comput. Sci.*, 366(3):194–198, Nov. 2006.

[3] N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proc. Am. Math. Soc.*, 16(1):109–114, 1965.

[4] M. Lothaire. *Combinatorics on Words*, volume 17 of *Encyclopaedia of Mathematics and its Applications*. Addison-Wesley, Reading, Mass., 1983. Reprinted in the *Cambridge Mathematical Library*, Cambridge University Press, Cambridge UK, 1997.

[5] M. Lothaire. *Algebraic Combinatorics on Words*. Number 90 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2002.

[6] M. Lothaire. *Applied Combinatorics on Words*. Number 105 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2005.

[7] R. C. Lyndon and P. Schützenberger. The equation $a^m = b^n c^p$ in a free group. *Michigan Math. J.*, 9:289–298, 1962.