Combinatorics on Words formalized
Graph Lemma

Štěpán Holub
Štěpán Starosta

October 13, 2025

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

Contents

theory *Glued-Codes*
imports *Combinatorics-Words.Submonoids*
begin

Chapter 1

Glued codes

1.1 Lists that do not end with a fixed letter

lemma *append-last-neq*:
$us = \varepsilon \lor last\ us \neq w \Longrightarrow vs = \varepsilon \lor last\ vs \neq w \Longrightarrow us \cdot vs = \varepsilon \lor last\ (us \cdot vs) \neq w$
by (*auto simp only*: *last-append split*: *if-split*)

lemma *last-neq-induct* [*consumes 1*, *case-names emp hd-eq hd-neq*]:
assumes *invariant*: $us = \varepsilon \lor last\ us \neq w$
and *emp*: $P\ \varepsilon$
and *hd-eq*: $\bigwedge us.\ us \neq \varepsilon \Longrightarrow last\ us \neq w \Longrightarrow P\ us \Longrightarrow P\ (w\ \#\ us)$
and *hd-neq*: $\bigwedge u\ us.\ u \neq w \Longrightarrow us = \varepsilon \lor last\ us \neq w \Longrightarrow P\ us \Longrightarrow P\ (u\ \#\ us)$
shows $P\ us$
using *invariant* **proof** (*induction us*)
case (*Cons u us*)
have *inv*: $us = \varepsilon \lor last\ us \neq w$
using *Cons.prems* **by** (*intro disjI*) *simp*
show $P\ (u\ \#\ us)$
proof (*cases*)
assume $u = w$
have $*$: $us \neq \varepsilon$ **and** $last\ us \neq w$
using *Cons.prems* **unfolding** ‹$u = w$› **by** *auto*
then show $P\ (u\ \#\ us)$ **unfolding** ‹$u = w$› **using** *Cons.IH*[*OF inv*] **by** (*fact hd-eq*)
qed (*use inv Cons.IH*[*OF inv*] **in** ‹*fact hd-neq*›)
qed (*rule* ‹$P\ \varepsilon$›)

lemma *last-neq-blockE*:
assumes *last-neq*: $us \neq \varepsilon$ **and** $last\ us \neq w$
obtains $k\ u\ us'$ **where** $u \neq w$ **and** $us' = \varepsilon \lor last\ us' \neq w$ **and** $[w]^{@}\ k \cdot u\ \#\ us' = us$
using *disjI2*[*OF* ‹$last\ us \neq w$›] ‹$us \neq \varepsilon$› **proof** (*induction us rule*: *last-neq-induct*)
case (*hd-eq us*)

from ‹*us* ≠ ε› **show** *?case*
by (*rule hd-eq.IH*[*rotated*]) (*intro hd-eq.prems*(*1*)[*of - - Suc -*], *assumption+*, *simp*)
next
case (*hd-neq u us*)
from *hd-neq.hyps* **show** *?case*
by (*rule hd-neq.prems*(*1*)[*of - - 0*]) *simp*
qed *blast*

lemma *last-neq-block-induct* [*consumes 1*, *case-names emp block*]:
assumes *last-neq*: *us* = ε ∨ *last us* ≠ *w*
and *emp*: *P* ε
and *block*: ⋀*k u us*. *u* ≠ *w* ⟹ *us* = ε ∨ *last us* ≠ *w* ⟹ *P us* ⟹ *P* ([*w*] @ *k* · (*u* # *us*))
shows *P us*
using *last-neq* **proof** (*induction us rule*: *ssuf-induct*)
case (*ssuf us*)
show *?case* **proof** (*cases us* = ε)
assume *us* ≠ ε
obtain *k u us'* **where** *u* ≠ *w* **and** *us'* = ε ∨ *last us'* ≠ *w* **and** [*w*] @ *k* · *u* # *us'* = *us*
using ‹*us* ≠ ε› ‹*us* = ε ∨ *last us* ≠ *w*› **by** (*elim last-neq-blockE*) (*simp add*: ‹*us* ≠ ε›)
have *us'* <*s us* **and** *us'* = ε ∨ *last us'* ≠ *w*
using ‹*us* = ε ∨ *last us* ≠ *w*› **by** (*auto simp flip*: ‹[*w*] @ *k* · *u* # *us'* = *us*›)
from ‹*u* ≠ *w*› ‹*us'* = ε ∨ *last us'* ≠ *w*› *ssuf.IH*[*OF this*]
show *P us* **unfolding** ‹[*w*] @ *k* · *u* # *us'* = *us*›[*symmetric*] **by** (*fact block*)
qed (*simp only*: *emp*)
qed

1.2 Glue a list element with its successors/predecessors

function *glue* :: *'a list* ⇒ *'a list list* ⇒ *'a list list* **where**
glue-emp: *glue w* ε = ε |
glue-Cons: *glue w* (*u* # *us*) =
(*let glue-tl* = *glue w us in*
if u = *w then* (*u* · *hd glue-tl*) # *tl glue-tl*
else u # *glue-tl*)
unfolding *prod-eq-iff prod.sel* **by** (*cases rule*: *list.exhaust*[*of snd -*]) *blast+*
termination by (*relation measure* (*length* ∘ *snd*)) *simp-all*

lemma *no-gluing*: *w* ∉ *set us* ⟹ *glue w us* = *us*
by (*induction us*) *auto*

lemma *glue-nemp* [*simp*, *intro*!]: *us* ≠ ε ⟹ *glue w us* ≠ ε
by (*elim hd-tlE*) (*auto simp only*: *glue.simps Let-def split*!: *if-split*)

lemma *glue-is-emp-iff* [*simp*]: *glue w us* = ε ⟷ *us* = ε
  using *glue-nemp glue-emp* **by** *blast*

lemma *len-glue*: *us* = ε ∨ *last us* ≠ *w* ⟹ |*glue w us*| + *count-list us w* = |*us*|
  by (*induction rule*: *last-neq-induct*) (*auto simp add*: *Let-def*)

lemma *len-glue-le*: **assumes** *us* = ε ∨ *last us* ≠ *w* **shows** |*glue w us*| ≤ |*us*|
  using *len-glue*[*OF assms*] **unfolding** *nat-le-iff-add eq-commute*[*of* |*us*|] **by** *blast*

lemma *len-glue-less* []: *us* = ε ∨ *last us* ≠ *w* ⟹ *w* ∈ *set us* ⟹ |*glue w us*| < |*us*|
  by (*simp add*: *count-list-gr-0-iff flip*: *len-glue*[*of us*])

lemma **assumes** *us* = ε ∨ *last us* ≠ *w* **and** ε ∉ *set us*
  shows *emp-not-in-glue*: ε ∉ *set* (*glue w us*)
    and *glued-not-in-glue*: *w* ∉ *set* (*glue w us*)
  unfolding *atomize-conj* **using** *assms* **by** (*induction us rule*: *last-neq-induct*)
    (*auto simp*: *Let-def dest*!: *tl-set lists-hd-in-set*[*OF glue-nemp*[*of - w*]])

lemma *glue-glue*: *us* = ε ∨ *last us* ≠ *w* ⟹ ε ∉ *set us* ⟹ *glue w* (*glue w us*) = *glue w us*
  using *no-gluing*[*OF glued-not-in-glue*]**.**

lemma *glue-block-append*: **assumes** *u* ≠ *w*
  shows *glue w* ([*w*] $^{@}$ *k* · (*u* # *us*)) = (*w* $^{@}$ *k* · *u*) # *glue w us*
  by (*induction k*) (*simp-all add*: ‹*u* ≠ *w*›)

lemma *concat-glue* [*simp*]: *us* = ε ∨ *last us* ≠ *w* ⟹ *concat* (*glue w us*) = *concat us*
  by (*induction us rule*: *last-neq-block-induct*) (*simp-all add*: *glue-block-append*)

lemma *glue-append*:
  us = ε ∨ *last us* ≠ *w* ⟹ *glue w* (*us* · *vs*) = *glue w us* · *glue w vs*
  by (*induction us rule*: *last-neq-block-induct*) (*simp-all add*: *glue-block-append*)

lemma *glue-pow*:
  assumes *us* = ε ∨ *last us* ≠ *w*
  shows *glue w* (*us* $^{@}$ *k*) = (*glue w us*) $^{@}$ *k*
  by (*induction k*) (*simp-all add*: *assms glue-append*)

lemma *glue-in-lists-hull* [*intro*]:
  us = ε ∨ *last us* ≠ *w* ⟹ *us* ∈ *lists G* ⟹ *glue w us* ∈ *lists* ⟨*G*⟩
 by (*induction rule*: *last-neq-induct*) (*simp-all add*: *Let-def tl-in-lists prod-cl gen-in*)

— Gluing from the right (gluing a letter with its predecessor)
function *gluer* :: *'a list* ⇒ *'a list list* ⇒ *'a list list* **where**
  gluer-emp: *gluer w* ε = ε |
  gluer-Cons: *gluer w* (*u* # *us*) =
    (*let gluer-butlast* = *gluer w* (*butlast* (*u* # *us*)) *in*
      if last (*u* # *us*) = *w then* (*butlast gluer-butlast*) · [*last gluer-butlast* · *last* (*u*

us)]
    else gluer-butlast · [*last* (*u* # *us*)])
  unfolding *prod-eq-iff prod.sel* **by** (*cases rule*: *list.exhaust*[*of snd* -]) *blast*+
  termination by (*relation measure* (*length* ∘ *snd*)) *simp-all*

lemma *gluer-nemp-def*: **assumes** *us* ≠ ε
  shows *gluer w us* =
   (*let gluer-butlast* = *gluer w* (*butlast us*) *in*
    if last us = *w then* (*butlast gluer-butlast*) · [*last gluer-butlast* · *last us*]
    else gluer-butlast · [*last us*])
  using *gluer-Cons*[*of w hd us tl us*] **unfolding** *hd-Cons-tl*[*OF* ‹*us* ≠ ε›]**.**

lemma *gluer-nemp*: **assumes** *us* ≠ ε **shows** *gluer w us* ≠ ε
  unfolding *gluer-nemp-def*[*OF* ‹*us* ≠ ε›]
  by (*simp only*: *Let-def split*!: *if-split*)

lemma *hd-neq-induct* [*consumes 1*, *case-names emp snoc-eq snoc-neq*]:
  assumes *invariant*: *us* = ε ∨ *hd us* ≠ *w*
   and *emp*: *P* ε
   and *snoc-eq*: ⋀*us*. *us* ≠ ε ⟹ *hd us* ≠ *w* ⟹ *P us* ⟹ *P* (*us* · [*w*])
   and *snoc-neq*: ⋀*u us*. *u* ≠ *w* ⟹ *us* = ε ∨ *hd us* ≠ *w* ⟹ *P us* ⟹ *P* (*us* ·
[*u*])
  shows *P us*
using *last-neq-induct*[**where** *P*=λ*x*. *P* (*rev x*) **for** *P*, *reversed*, *unfolded rev-rev-ident*,
OF assms]**.**

lemma *gluer-rev* [*reversal-rule*]: **assumes** *us* = ε ∨ *last us* ≠ *w*
  shows *gluer* (*rev w*) (*rev* (*map rev us*)) = *rev* (*map rev* (*glue w us*))
  using *assms* **by** (*induction us rule*: *last-neq-induct*)
   (*simp-all add*: *gluer-nemp-def Let-def map-tl last-rev hd-map*)

lemma *glue-rev* [*reversal-rule*]: **assumes** *us* = ε ∨ *hd us* ≠ *w*
  shows *glue* (*rev w*) (*rev* (*map rev us*)) = *rev* (*map rev* (*gluer w us*))
  using *assms* **by** (*induction us rule*: *hd-neq-induct*)
   (*simp-all add*: *gluer-nemp-def Let-def map-tl last-rev hd-map*)

1.3 Generators with glued element

The following set will turn out to be the generating set of all words whose decomposition into a generating code does not end with w

inductive-set *glued-gens* :: *'a list* ⇒ *'a list set* ⇒ *'a list set*
  for *w G* **where**
   other-gen: *g* ∈ *G* ⟹ *g* ≠ *w* ⟹ *g* ∈ *glued-gens w G*
  | *glued* [*intro*!]: *u* ∈ *glued-gens w G* ⟹ *w* · *u* ∈ *glued-gens w G*

lemma *in-glued-gensI*: **assumes** *g* ∈ *G g* ≠ *w*
  shows $w^{@}$ *k* · *g* = *u* ⟹ *u* ∈ *glued-gens w G*
  by (*induction k arbitrary*: *u*) (*auto simp*: *other-gen*[*OF* ‹*g* ∈ *G*› ‹*g* ≠ *w*›])

lemma *in-glued-gensE*:
  assumes *u* ∈ *glued-gens w G*
  obtains *k g* **where** *g* ∈ *G* **and** *g* ≠ *w* **and** *w* @ *k* · *g* = *u*
using *assms* **proof** (*induction*)
  case (*glued u*)
    show *?case* **by** (*auto intro*!: *glued.IH*[*OF glued.prems*[*of* - *Suc* -]])
qed (*use pow-zero* **in** *blast*)

lemma *glued-gens-alt-def*: *glued-gens w C* = {*w* @ *k* · *g* | *k g*. *g* ∈ *C* ∧ *g* ≠ *w*}
  by (*blast elim*!: *in-glued-gensE intro*: *in-glued-gensI*)

lemma *glued-hull-sub-hull* [*simp*, *intro*!]: *w* ∈ *G* ⟹ ⟨*glued-gens w G*⟩ ⊆ ⟨*G*⟩
  by (*rule hull-mono′*) (*auto elim*!: *in-glued-gensE*)

lemma *glued-hull-sub-hull′*: *w* ∈ *G* ⟹ *u* ∈ ⟨*glued-gens w G*⟩ ⟹ *u* ∈ ⟨*G*⟩
  using *set-mp*[*OF glued-hull-sub-hull*]**.**

lemma *in-glued-hullE*:
  assumes *w* ∈ *G* **and** *u* ∈ ⟨*glued-gens w G*⟩
  obtains *us* **where** *concat us* = *u* **and** *us* ∈ *lists G* **and** *us* = *ε* ∨ *last us* ≠ *w*
using ‹*u* ∈ ⟨*glued-gens w G*⟩› **proof** (*induction arbitrary*: *thesis*)
  case (*prod-cl v u*)
    obtain *k g* **where** *g* ∈ *G* **and** *g* ≠ *w* **and** *concat* ([*w*] @ *k* · [*g*]) = *v*
      using ‹*v* ∈ *glued-gens w G*› **by** *simp* (*elim in-glued-gensE*)
    obtain *us* **where** *u*: *concat us* = *u* **and** *us* ∈ *lists G* **and** (*us* = *ε* ∨ *last us* ≠ *w*) **by** *fact*
    have *concat* ([*w*] @ *k* · [*g*] · *us*) = *v* · *u*
      by (*simp flip*: ‹*concat* ([*w*] @ *k* · [*g*]) = *v*› ‹*concat us* = *u*›)
    with ‹(*us* = *ε* ∨ *last us* ≠ *w*)› **show** *thesis*
      by (*elim prod-cl.prems*, *intro lists.intros*
          append-in-lists pow-in-lists ‹*w* ∈ *G*› ‹*g* ∈ *G*› ‹*us* ∈ *lists G*›)
        (*auto simp*: ‹*g* ≠ *w*›)
qed (*use concat.simps*(*1*) **in** *blast*)

lemma *glue-in-lists* [*simp*, *intro*!]:
  assumes *us* = *ε* ∨ *last us* ≠ *w*
  shows *us* ∈ *lists G* ⟹ *glue w us* ∈ *lists* (*glued-gens w G*)
  using *assms* **by** (*induction rule*: *last-neq-block-induct*)
    (*auto simp*: *glue-block-append intro*: *in-glued-gensI*)

lemma *concat-in-glued-hull*[*intro*]:
  us ∈ *lists G* ⟹ *us* = *ε* ∨ *last us* ≠ *w* ⟹ *concat us* ∈ ⟨*glued-gens w G*⟩
  unfolding *concat-glue*[*symmetric*] **by** (*intro concat-in-hull′ glue-in-lists*)

lemma *glued-hull-conv*: **assumes** *w* ∈ *G*
  shows ⟨*glued-gens w G*⟩ = {*concat us* | *us*. *us* ∈ *lists G* ∧ (*us* = *ε* ∨ *last us* ≠ *w*)}
  by (*blast elim*!: *in-glued-hullE*[*OF* ‹*w* ∈ *G*›])

1.4 Bounded gluing

lemma *bounded-glue-in-lists*:
  assumes $us = \varepsilon \lor last\ us \neq w$ **and** $\neg\ [w]\ ^{@}\ n \leq f\ us$
  shows $us \in lists\ G \Longrightarrow glue\ w\ us \in lists\ \{w\ ^{@}\ k \cdot g \mid k\ g.\ g \in G \land g \neq w \land k < n\}$
using *assms* **proof** (*induction us rule*: *last-neq-block-induct*)
  case (*block k u us*)
    have $k < n$ **and** $\neg\ [w]\ ^{@}\ n \leq f\ us$
      using ‹$\neg\ [w]\ ^{@}\ n \leq f\ [w]\ ^{@}\ k \cdot u \ \#\ us$›
      by (*blast intro*!: *not-le-imp-less*, *blast intro*!: *fac-ext-pref fac-ext-hd*)
    then show *?case*
      using ‹$[w]\ ^{@}\ k \cdot u \ \#\ us \in lists\ G$› ‹$u \neq w$› **unfolding** *glue-block-append*[*OF* ‹$u \neq w$›]
      by (*blast intro*!: *block.IH del*: *in-listsD in-listsI*)
qed *simp*

1.4.1 Gluing on binary alphabet

lemma *bounded-bin-glue-in-lists*: — meaning: a binary code
  assumes $us = \varepsilon \lor last\ us \neq x$
      and $\neg\ [x]\ ^{@}\ n \leq f\ us$
      and $us \in lists\ \{x,\ y\}$
  shows $glue\ x\ us \in lists\ \{x\ ^{@}\ k \cdot y \mid k.\ k < n\}$
using *bounded-glue-in-lists*[*OF assms*] **by** *blast*

lemma *single-bin-glue-in-lists*: — meaning: a single occurrence
  assumes $us = \varepsilon \lor last\ us \neq x$
      and $\neg\ [x,x] \leq f\ us$
      and $us \in lists\ \{x,\ y\}$
  shows $glue\ x\ us \in lists\ \{x \cdot y,\ y\}$
  using *bounded-bin-glue-in-lists*[*of - - 2*, *simplified*, *OF assms*] **unfolding** *numeral-nat*
  by (*auto elim*!: *sub-lists-mono*[*rotated*] *less-SucE*)

lemma *count-list-single-bin-glue*:
  assumes $x \neq \varepsilon$ **and** $x \neq y$
      and $us = \varepsilon \lor last\ us \neq x$
      and $us \in lists\ \{x,y\}$
      and $\neg\ [x,x] \leq f\ us$
  shows $count\text{-}list\ (glue\ x\ us)\ (x \cdot y) = count\text{-}list\ us\ x$
    and $count\text{-}list\ (glue\ x\ us)\ y + count\text{-}list\ us\ x = count\text{-}list\ us\ y$
using *assms*(*3–5*) **unfolding** *atomize-conj pow-Suc*[*symmetric*]
proof (*induction us rule*: *last-neq-block-induct*)
  case (*block k u us*)
    have $u = y$ **using** ‹$[x]\ ^{@}\ k \cdot u \ \#\ us \in lists\ \{x,\ y\}$› ‹$u \neq x$› **by** *simp*
    have *IH*: $count\text{-}list\ (glue\ x\ us)\ (x \cdot y) = count\text{-}list\ us\ x \land$
          $count\text{-}list\ (glue\ x\ us)\ y + count\text{-}list\ us\ x = count\text{-}list\ us\ y$
    using *block.prems* **by** (*intro block.IH*) (*simp*, *blast intro*!: *fac-ext-pref fac-ext-hd*)
    have $\neg\ [x]\ ^{@}\ Suc\ (Suc\ 0) \leq f\ [x]\ ^{@}\ k \cdot u \ \#\ us$

using *block.prems*(*2*) **by** *auto*
then have $k < Suc\ (Suc\ 0)$
by (*blast intro*!: *not-le-imp-less*)
then show *?case* **unfolding** ‹$u = y$› *glue-block-append*[*OF* ‹$x \neq y$›[*symmetric*]]
by (*elim less-SucE less-zeroE*) (*simp-all add*: ‹$x \neq y$› ‹$x \neq y$›[*symmetric*] ‹$x \neq \varepsilon$› *IH*)
qed *simp*

1.5 Code with glued element

context *code*
begin

If the original generating set is a code, then also the glued generators form a code

lemma *glued-hull-last-dec*: **assumes** $w \in \mathcal{C}$ **and** $u \in \langle glued\text{-}gens\ w\ \mathcal{C}\rangle$ **and** $u \neq \varepsilon$
shows *last* (*Dec* $\mathcal{C}$ *u*) $\neq w$
using ‹$u \in \langle glued\text{-}gens\ w\ \mathcal{C}\rangle$›
by (*elim in-glued-hullE*[*OF* ‹$w \in \mathcal{C}$›]) (*auto simp*: *code-unique-dec* ‹$u \neq \varepsilon$›)

lemma *in-glued-hullI* [*intro*]:
assumes $u \in \langle\mathcal{C}\rangle$ **and** ($u = \varepsilon \vee$ *last* (*Dec* $\mathcal{C}$ *u*) $\neq w$)
shows $u \in \langle glued\text{-}gens\ w\ \mathcal{C}\rangle$
using *concat-in-glued-hull*[*OF dec-in-lists*[*OF* ‹$u \in \langle\mathcal{C}\rangle$›], *of w*]
by (*simp add*: ‹$u \in \langle\mathcal{C}\rangle$› ‹$u = \varepsilon \vee$ *last* (*Dec* $\mathcal{C}$ *u*) $\neq w$›)

lemma *code-glued-hull-conv*: **assumes** $w \in \mathcal{C}$
shows $\langle glued\text{-}gens\ w\ \mathcal{C}\rangle = \{u \in \langle\mathcal{C}\rangle.\ u = \varepsilon \vee last\ (Dec\ \mathcal{C}\ u) \neq w\}$
proof
show $\langle glued\text{-}gens\ w\ \mathcal{C}\rangle \subseteq \{u \in \langle\mathcal{C}\rangle.\ u = \varepsilon \vee last\ (Dec\ \mathcal{C}\ u) \neq w\}$
using *glued-hull-sub-hull'*[*OF* ‹$w \in \mathcal{C}$›] *glued-hull-last-dec*[*OF* ‹$w \in \mathcal{C}$›] **by** *blast*
show $\{u \in \langle\mathcal{C}\rangle.\ u = \varepsilon \vee last\ (Dec\ \mathcal{C}\ u) \neq w\} \subseteq \langle glued\text{-}gens\ w\ \mathcal{C}\rangle$
using *in-glued-hullI* **by** *blast*
qed

lemma *in-glued-hull-iff*:
assumes $w \in \mathcal{C}$ **and** $u \in \langle\mathcal{C}\rangle$
shows $u \in \langle glued\text{-}gens\ w\ \mathcal{C}\rangle \longleftrightarrow u = \varepsilon \vee last\ (Dec\ \mathcal{C}\ u) \neq w$
by (*simp add*: ‹$w \in \mathcal{C}$› ‹$u \in \langle\mathcal{C}\rangle$› *code-glued-hull-conv*)

lemma *glued-not-in-glued-hull*: $w \in \mathcal{C} \Longrightarrow w \notin \langle glued\text{-}gens\ w\ \mathcal{C}\rangle$
unfolding *in-glued-hull-iff*[*OF* - *gen-in*] *code-el-dec*
by (*simp add*: *nemp*)

lemma *glued-gens-nemp*: **assumes** $u \in glued\text{-}gens\ w\ \mathcal{C}$ **shows** $u \neq \varepsilon$
using *assms* **by** (*induction*) (*auto simp add*: *nemp*)

lemma *glued-gens-code*: **assumes** $w \in \mathcal{C}$ **shows** *code* (*glued-gens w* $\mathcal{C}$)
proof

show *us* = *vs* **if** *us* ∈ *lists* (*glued-gens w* 𝒞) **and** *vs* ∈ *lists* (*glued-gens w* 𝒞)
and *concat us* = *concat vs* **for** *us vs*
using *that* **proof** (*induction rule*: *list-induct2′*)
case (*4 u us v vs*)
have ∗: *us* ∈ *lists* (*glued-gens w* 𝒞) ⟹ *us* ∈ *lists* ⟨𝒞⟩ **for** *us*
using *sub-lists-mono*[*OF subset-trans*[*OF genset-sub glued-hull-sub-hull*[*OF* ‹*w* ∈ 𝒞›]]]**.**
obtain *k u′ l v′*
where *u′* ∈ 𝒞 *u′* ≠ *w w* $^{@}$ *k* · *u′* = *u*
and *v′* ∈ 𝒞 *v′* ≠ *w w* $^{@}$ *l* · *v′* = *v*
using *4.prems*(*1*−*2*) **by** *simp* (*elim conjE in-glued-gensE*)
from *this*(*3*, *6*) *4.prems* ‹*w* ∈ 𝒞›
have *concat* (([*w*] $^{@}$ *k* · [*u′*]) · (*Ref* 𝒞 *us*)) = *concat* (([*w*] $^{@}$ *l* · [*v′*]) · (*Ref* 𝒞 *vs*))
by (*simp add*: *concat-ref* ∗ *lassoc*)
with ‹*w* ∈ 𝒞› ‹*u′* ∈ 𝒞› ‹*v′* ∈ 𝒞› *4.prems*(*1*−*2*)
have [*w*] $^{@}$ *k* · [*u′*] ⋈ [*w*] $^{@}$ *l* · [*v′*]
by (*elim eqd-comp*[*OF is-code*, *rotated 2*])
(*simp-all add*: ∗ *pow-in-lists ref-in′*)
with ‹*u′* ≠ *w*› ‹*v′* ≠ *w*› ‹*w* $^{@}$ *k* · *u′* = *u*› ‹*w* $^{@}$ *l* · *v′* = *v*›
have *u* = *v*
by (*elim sing-pref-comp-mismatch*[*rotated 2*, *elim-format*]) *blast*+
then show *u* # *us* = *v* # *vs*
using *4.IH 4.prems*(*1*−*3*) **by** *simp*
qed (*auto dest*!: *glued-gens-nemp*)
qed

A crucial lemma showing the relation between gluing and the decomposition into generators

lemma *dec-glued-gens*: **assumes** *w* ∈ 𝒞 **and** *u* ∈ ⟨*glued-gens w* 𝒞⟩
shows *Dec* (*glued-gens w* 𝒞) *u* = *glue w* (*Dec* 𝒞 *u*)
using ‹*u* ∈ ⟨*glued-gens w* 𝒞⟩› *glued-hull-sub-hull′*[*OF* ‹*w* ∈ 𝒞› ‹*u* ∈ ⟨*glued-gens w* 𝒞⟩›]
by (*intro code.code-unique-dec glued-gens-code*)
(*simp-all add*: *in-glued-hull-iff* ‹*w* ∈ 𝒞›)

lemma *ref-glue*: *us* = ε ∨ *last us* ≠ *w* ⟹ *us* ∈ *lists* 𝒞 ⟹ *Ref* 𝒞 (*glue w us*) = *us*
by (*intro refI glue-in-lists-hull*) *simp-all*

end
theorem *glued-code-right*:
assumes *code C* **and** *w* ∈ *C*
shows *code* {*w* $^{@}$ *k* · *u* |*k u*. *u* ∈ *C* ∧ *u* ≠ *w*}
using *code.glued-gens-code*[*OF* ‹*code C*› ‹*w* ∈ *C*›] **unfolding** *glued-gens-alt-def***.**

theorem *glued-code*:
assumes *code C* **and** *w* ∈ *C*
shows *code* {*u* · *w* $^{@}$ *k* |*k u*. *u* ∈ *C* ∧ *u* ≠ *w*}
using *glued-code-right*[*reversed*, *OF assms*]**.**

1.6 Gluing is primitivity preserving

It is easy to obtain that gluing lists of code elements preserves primitivity. We provide the result under weaker condition where glue blocks of the list have unique concatenation.

lemma (**in** *code*) *code-prim-glue*:
  assumes *last-neq*: $us = \varepsilon \vee last\ us \neq w$
    and $us \in lists\ \mathcal{C}$
  shows $primitive\ us \Longrightarrow primitive\ (glue\ w\ us)$
  using *prim-map-prim*[*OF prim-concat-prim*, *of decompose* $\mathcal{C}$ *glue w us*]
  unfolding *refine-def*[*symmetric*] *ref-glue*[*OF assms*].

— In the context of code the inverse to the glue function is the *refine* function, i.e. $\lambda vs.\ concat\ (map\ (decompose\ \mathcal{C})\ vs)$, see $[\![code\ ?\mathcal{C};\ ?us = \varepsilon \vee last\ ?us \neq ?w;\ ?us \in lists\ ?\mathcal{C}]\!] \Longrightarrow Ref\ ?\mathcal{C}\ glue\ ?w\ ?us = ?us$. The role of the *decompose* function outside the code context supply the 'unglue' function, which maps glued blocks to its unique preimages (see below).

definition *glue-block* :: $'a\ list \Rightarrow 'a\ list\ list \Rightarrow\ 'a\ list\ list \Rightarrow bool$
  where $glue\text{-}block\ w\ us\ bs =$
    $(\exists\, ps\ k\ u\ ss.\ (ps = \varepsilon \vee last\ ps \neq w) \wedge u \neq w \wedge ps \cdot [w]\ ^{@}\ k \cdot u \# ss = us \wedge [w]\ ^{@}\ k \cdot [u] = bs)$

lemma *glue-blockI* [*intro*]:
  $ps = \varepsilon \vee last\ ps \neq w \Longrightarrow u \neq w \Longrightarrow ps \cdot [w]\ ^{@}\ k \cdot u \# ss = us \Longrightarrow [w]\ ^{@}\ k \cdot [u] = bs$
    $\Longrightarrow glue\text{-}block\ w\ us\ bs$
  unfolding *glue-block-def* **by** (*intro exI conjI*)

lemma *glue-blockE*:
  assumes *glue-block w us bs*
  obtains *ps k u ss* **where** $ps = \varepsilon \vee last\ ps \neq w$ **and** $u \neq w\ ps \cdot [w]\ ^{@}\ k \cdot u \# ss = us$
    and $[w]\ ^{@}\ k \cdot [u] = bs$
  using *assms* **unfolding** *glue-block-def* **by** (*elim exE conjE*)

lemma assumes *glue-block w us bs*
  shows *glue-block-of-appendL*: $glue\text{-}block\ w\ (us \cdot vs)\ bs$
   and *glue-block-of-appendR*: $vs = \varepsilon \vee last\ vs \neq w \Longrightarrow glue\text{-}block\ w\ (vs \cdot us)\ bs$
  using ‹*glue-block w us bs*› **by** (*elim glue-blockE*, *use nothing* **in** ‹
   intro glue-blockI[*of - w - - - -* · *vs us* · *vs bs*]
     glue-blockI[*OF append-last-neq*, *of vs w - - - - vs* · *us bs*],
   simp-all only: *eq-commute*[*of - us*] *rassoc append-Cons refl not-False-eq-True*›)+

lemma *glue-block-of-block-append*:
  $u \neq w \Longrightarrow glue\text{-}block\ w\ us\ bs \Longrightarrow glue\text{-}block\ w\ ([w]\ ^{@}\ k \cdot u \# us)\ bs$
  by (*simp only*: *hd-word*[*of - us*] *lassoc*) (*elim glue-block-of-appendR*, *simp-all*)

lemma *in-set-glueE*:
assumes *last-neq*: $us = \varepsilon \vee last\ us \neq w$
and $b \in set\ (glue\ w\ us)$
obtains *bs* **where** *glue-block w us bs* **and** *concat bs = b*
using *assms* **proof** (*induction us rule*: *last-neq-block-induct*)
case (*block k u us*)
show *thesis* **using** ‹$b \in set\ (glue\ w\ ([w]\ ^{@}\ k \cdot u\ \#\ us))$›
proof (*auto simp add*: *glue-block-append* ‹$u \neq w$›)
show $b = w\ ^{@}\ k \cdot u \Longrightarrow thesis$
by (*auto intro*!: *block.prems*(*1*) *glue-blockI*[*OF* - ‹$u \neq w$› - *refl*])
show $b \in set\ (glue\ w\ us) \Longrightarrow thesis$
by (*auto intro*!: *block.IH*[*OF block.prems*(*1*)] *glue-block-of-block-append* ‹u
$\neq w$›)
qed
qed *simp*

definition *unglue* :: *'a list* ⇒ *'a list list* ⇒ *'a list* ⇒ *'a list list*
where *unglue w us b* = (*THE bs. glue-block w us bs* ∧ *concat bs* = *b*)

lemma *unglueI*:
assumes *unique-blocks*: $\bigwedge bs_1\ bs_2.\ glue\text{-}block\ w\ us\ bs_1 \Longrightarrow glue\text{-}block\ w\ us\ bs_2$
$\Longrightarrow concat\ bs_1 = concat\ bs_2 \Longrightarrow bs_1 = bs_2$
shows *glue-block w us bs* ⟹ *concat bs* = *b* ⟹ *unglue w us b* = *bs*
unfolding *unglue-def* **by** (*blast intro*: *unique-blocks*)

lemma *concat-map-unglue-glue*:
assumes *last-neq*: $us = \varepsilon \vee last\ us \neq w$
and *unique-blocks*: $\bigwedge vs_1\ vs_2.\ glue\text{-}block\ w\ us\ vs_1 \Longrightarrow glue\text{-}block\ w\ us\ vs_2$
$\Longrightarrow concat\ vs_1 = concat\ vs_2 \Longrightarrow vs_1 = vs_2$
shows *concat* (*map* (*unglue w us*) (*glue w us*)) = *us*
using *assms* **proof** (*induction us rule*: *last-neq-block-induct*)
case (*block k u us*)
have *IH*: *concat* (*map* (*unglue w us*) (*glue w us*)) = *us*
using *block.IH*[*OF block.prems*] **by** (*blast intro*!: *glue-block-of-block-append* ‹u
$\neq w$›)
have ∗: *map* (*unglue w* ($[w]\ ^{@}\ k \cdot u\ \#\ us$)) (*glue w us*) = *map* (*unglue w us*)
(*glue w us*)
by (*auto simp only*: *map-eq-conv unglue-def del*: *the-equality*
elim!: *in-set-glueE*[*OF* ‹$us = \varepsilon \vee last\ us \neq w$›], *intro the-equality*)
(*simp-all only*: *the-equality block.prems glue-block-of-block-append*[*OF* ‹$u \neq$
w›])
show *concat* (*map* (*unglue w* ($[w]\ ^{@}\ k \cdot u\ \#\ us$)) (*glue w* ($[w]\ ^{@}\ k \cdot u\ \#\ us$)))
$= [w]\ ^{@}\ k \cdot u\ \#\ us$
by (*auto simp add*: *glue-block-append*[*OF* ‹$u \neq w$›] ∗ *IH*
intro!: *unglueI intro*: *glue-blockI*[*OF* - ‹$u \neq w$›] *block.prems*)
qed *simp*

lemma *prim-glue*:
assumes *last-neq*: $us = \varepsilon \vee last\ us \neq w$

and *unique-blocks*: $\bigwedge bs_1\ bs_2.\ glue\text{-}block\ w\ us\ bs_1 \Longrightarrow glue\text{-}block\ w\ us\ bs_2$
$\Longrightarrow concat\ bs_1 = concat\ bs_2 \Longrightarrow bs_1 = bs_2$
shows *primitive us* $\Longrightarrow$ *primitive* (*glue w us*)
using *prim-map-prim*[*OF prim-concat-prim*, *of unglue w us glue w us*]
by (*simp only*: *concat-map-unglue-glue assms*)

1.6.1 Gluing on binary alphabet

lemma *bin-glue-blockE*:
assumes *us* $\in$ *lists* $\{x,\ y\}$
and *glue-block x us bs*
obtains k **where** $[x]\ ^{@}\ k \cdot [y] = bs$
using *assms* **by** (*auto simp only*: *glue-block-def del*: *in-listsD*)

lemma *unique-bin-glue-blocks*:
assumes *us* $\in$ *lists* $\{x,\ y\}$ **and** $x \neq \varepsilon$
shows $glue\text{-}block\ x\ us\ bs_1 \Longrightarrow glue\text{-}block\ x\ us\ bs_2 \Longrightarrow concat\ bs_1 = concat\ bs_2 \Longrightarrow bs_1 = bs_2$
by (*auto simp*: *eq-pow-exp*[*OF* ‹$x \neq \varepsilon$›] *elim*!: *bin-glue-blockE*[*OF* ‹*us* $\in$ *lists* $\{x,\ y\}$›])

lemma *prim-bin-glue*:
assumes *us* $\in$ *lists* $\{x,\ y\}$ **and** $x \neq \varepsilon$
and $us = \varepsilon \lor last\ us \neq x$
shows *primitive us* $\Longrightarrow$ *primitive* (*glue x us*)
using *prim-glue*[*OF* ‹$us = \varepsilon \lor last\ us \neq x$› *unique-bin-glue-blocks*[*OF assms*(*1*−*2*)]]**.**

end

theory *Graph-Lemma*
imports *Combinatorics-Words.Submonoids Glued-Codes*

begin

Chapter 2

Graph Lemma

The Graph Lemma is an important tool for gaining information about systems of word equations. It yields an upper bound on the rank of the solution, that is, on the number of factors into all images of unknowns can be factorized. The most straightforward application is showing that a system of equations admits periodic solutions only, which in particular holds for any nontrivial equation over two words.

The name refers to a graph whose vertices are the unknowns of the system, and edges connect front letters of the left- and right- hand sides of equations. The bound mentioned above is then the number of connected components of the graph.

We formalize the algebraic proof from [1]. Key ingredients of the proof are in the theory *Combinatorics-Words-Graph-Lemma.Glued-Codes*

2.1 Graph lemma

theorem *graph-lemma-last*: $\mathfrak{B}_F\ G = \{last\ (Dec\ (\mathfrak{B}_F\ G)\ g) \mid g.\ g \in G \land g \neq \varepsilon\}$
proof
  interpret *code* $\mathfrak{B}_F\ G$
    using *free-basis-code*.
  — the core is to show that each element of the free basis must be a last of some word
  show $\mathfrak{B}_F\ G \subseteq \{last\ (Dec\ \mathfrak{B}_F\ G\ g)\ |g.\ g \in G \land g \neq \varepsilon\}$
  proof (*rule ccontr*)
    — Assume the contrary.
    assume $\neg\ \mathfrak{B}_F\ G \subseteq \{last\ (Dec\ \mathfrak{B}_F\ G\ g)\ |g.\ g \in G \land g \neq \varepsilon\}$
    — And let w be the not-last
    then obtain *w*
      where $w \in \mathfrak{B}_F\ G$
        and *hd-dec-neq*: $\bigwedge g.\ g \in G \Longrightarrow g \neq \varepsilon \Longrightarrow last\ (Dec\ (\mathfrak{B}_F\ G)\ g) \neq w$
      by *blast*
    — For contradiction: We have a free hull which does not contain w but contains

G.
  have $G \subseteq \langle$*glued-gens w* $(\mathfrak{B}_F\ G)\rangle$
    by (*blast intro*!: *gen-in-free-hull hd-dec-neq del*: *notI*)
  then have $\langle\mathfrak{B}_F\ G\rangle \subseteq \langle$*glued-gens w* $(\mathfrak{B}_F\ G)\rangle$
    unfolding *basis-gen-hull-free*
    by (*intro code.free-hull-min glued-gens-code* ‹$w \in \mathfrak{B}_F\ G$›)
  then show *False*
    using ‹$w \in \mathfrak{B}_F\ G$› *glued-not-in-glued-hull* **by** *blast*
 qed
 — The opposite inclusion is easy
 show $\{$*last* (*Dec* $\mathfrak{B}_F\ G\ g) \mid g.\ g \in G \wedge g \neq \varepsilon\} \subseteq \mathfrak{B}_F\ G$
  by (*auto intro*!: *dec-in-lists lists-hd-in-set*[*reversed*] *gen-in-free-hull del*: *notI*)
qed

theorem *graph-lemma*: $\mathfrak{B}_F\ G = \{$*hd* (*Dec* $(\mathfrak{B}_F\ G)\ g) \mid g.\ g \in G \wedge g \neq \varepsilon\}$
proof –
 have $*$: *rev u* = *last* (*Dec rev* ‘ $(\mathfrak{B}_F\ G)$ (*rev g*)) $\wedge\ g \in G \wedge g \neq \varepsilon$
     $\longleftrightarrow u =$ *hd* (*Dec* $(\mathfrak{B}_F\ G)\ g) \wedge g \in G \wedge g \neq \varepsilon$ **for** *u g*
  by (*cases* $g \in G \wedge g \neq \varepsilon$) (*simp add*: *gen-in-free-hull last-rev hd-map code.dec-rev*, *blast*)
 show *?thesis*
  using *graph-lemma-last*[*reversed*, *of G*] **unfolding** $*$.
qed

2.2 Binary code

We illustrate the use of the Graph Lemma in an alternative proof of the fact that two non-commuting words form a code. See also $[\![u_0 \cdot u_1 \neq u_1 \cdot u_0$; *us* $\in$ *lists* $\{u_0, u_1\}$; *vs* $\in$ *lists* $\{u_0, u_1\}$; *concat us* = *concat vs*$]\!] \Longrightarrow$ *us* = *vs* in *Combinatorics-Words.CoWBasic*.

First, we prove a lemma which is the core of the alternative proof.

lemma *non-comm-hds-neq*: **assumes** $u \cdot v \neq v \cdot u$ **shows** *hd* (*Dec* $\mathfrak{B}_F\ \{u,v\}\ u)$ $\neq$ *hd* (*Dec* $\mathfrak{B}_F\ \{u,v\}\ v)$
using *assms* **proof** (*rule contrapos-nn*)
 assume *hds-eq*: *hd* (*Dec* $\mathfrak{B}_F\ \{u,v\}\ u)$ = *hd* (*Dec* $\mathfrak{B}_F\ \{u,v\}\ v)$
 have $**$: $\mathfrak{B}_F\ \{u,v\} = \{$*hd* (*Dec* $\mathfrak{B}_F\ \{u,v\}\ u)\}$
  using *graph-lemma* **by** (*rule trans*) (*use assms* **in** ‹*auto intro*: *hds-eq*[*symmetric*]›)
 show $u \cdot v = v \cdot u$
  by(*intro comm-rootI*[*of* - *hd* (*Dec* $\mathfrak{B}_F\ \{u,v\}\ u)$])
  (*simp-all add*: $**$[*symmetric*] *gen-in-free-hull*)
qed

theorem assumes $u \cdot v \neq v \cdot u$ **shows** *code* $\{u, v\}$
proof (*rule code.intro*)
 have $*$: $w \in \{u, v\} \Longrightarrow w \neq \varepsilon$ **for** *w*
  using ‹$u \cdot v \neq v \cdot u$› **by** *blast*
 fix *xs ys*

show *xs* ∈ *lists* {*u*, *v*} ⟹ *ys* ∈ *lists* {*u*, *v*} ⟹ *concat xs* = *concat ys* ⟹ *xs* = *ys*
proof (*induction xs ys rule*: *list-induct2′*)
case (*4 x xs y ys*)
have ∗∗: *hd* (*Dec* 𝔅*F* {*u*,*v*} (*concat* (*z* # *zs*))) = *hd* (*Dec* 𝔅*F* {*u*,*v*} *z*)
if *z* # *zs* ∈ *lists* {*u*, *v*} **for** *z zs*
using *that* **by** (*elim listsE*) (*simp del*: *insert-iff*
add: *concat-in-hull′ gen-in set-mp*[*OF hull-sub-free-hull*]
free-basis-dec-morph ∗ *basis-gen-hull-free*)
have *hd* (*Dec* 𝔅*F* {*u*,*v*} *x*) = *hd* (*Dec* 𝔅*F* {*u*,*v*} *y*)
using *4.prems* **by** (*simp only*: ∗∗[*symmetric*])
then have *x* = *y*
using *4.prems*(*1*−*2*) *non-comm-hds-neq*[*OF* ‹*u* · *v* ≠ *v* · *u*›]
by (*elim listsE insertE emptyE*) *simp-all*
with *4* **show** *x* # *xs* = *y* # *ys* **by** *simp*
qed (*simp-all add*: ∗)
qed

end

References

[1] J. Berstel, D. Perrin, J. Perrot, and A. Restivo. Sur le théorème du défaut. *Journal of Algebra*, 60(1):169–180, 1979.