Decreasing-Diagrams-II

By Bertram Felgenhauer

February 6, 2026

Abstract

This theory formalizes a commutation version of decreasing diagrams for Church-Rosser modulo. The proof follows Felgenhauer and van Oostrom (RTA 2013). The theory also provides important specializations, in particular van Oostrom's conversion version (TCS 2008) of decreasing diagrams.

We follow the development described in [1]: Conversions are mapped to Greek strings, and we prove that whenever a local peak (or cliff) is replaced by a joining sequence from a locally decreasing diagram, then the corresponding Greek strings become smaller in a specially crafted well-founded order on Greek strings. Once there are no more local peaks or cliffs are left, the result is a valley that establishes the Church-Rosser modulo property.

As special cases we provide non-commutation versions and the conversion version of decreasing diagrams by van Oostrom [3]. We also formalize extended decreasingness [2].

Contents

1 Preliminaries **2**
- 1.1 Trivialities . . . 2
- 1.2 Complete lattices and least fixed points . . . 2
  - 1.2.1 A chain-based induction principle . . . 2
  - 1.2.2 Preservation of transitivity, asymmetry, irreflexivity by suprema . . . 3
- 1.3 Multiset extension . . . 4
- 1.4 Incrementality of *mult1* and *mult* . . . 5
- 1.5 Well-orders and well-quasi-orders . . . 6
- 1.6 Splitting lists into prefix, element, and suffix . . . 6

2 Decreasing Diagrams **7**
- 2.1 Greek accents . . . 7
- 2.2 Comparing Greek strings . . . 8
- 2.3 Preservation of strict partial orders . . . 9

2.4 Involution 10
2.5 Monotonicity of *greek-less r* 12
2.6 Well-founded-ness of *greek-less r* 13
2.7 Basic Comparisons 14
2.8 Labeled abstract rewriting 19

3 Results **24**
3.1 Church-Rosser modulo 24
3.2 Commutation and confluence 27
3.3 Extended decreasing diagrams 27

1 Preliminaries

theory *Decreasing-Diagrams-II-Aux*
imports
Well-Quasi-Orders.Multiset-Extension
Well-Quasi-Orders.Well-Quasi-Orders
begin

1.1 Trivialities

abbreviation *strict-order R* ≡ *irrefl R* ∧ *trans R*

lemma *strict-order-strict*: *strict-order q* ⟹ *strict* (λ*a b*. (*a*, *b*) ∈ $q^{=}$) = (λ*a b*. (*a*, *b*) ∈ *q*)
unfolding *trans-def irrefl-def* **by** *fast*

lemma *mono-lex1*: *mono* (λ*r*. *lex-prod r s*)
by (*auto simp add*: *mono-def*)

lemma *mono-lex2*: *mono* (*lex-prod r*)
by (*auto simp add*: *mono-def*)

lemmas *converse-inward* = *rtrancl-converse*[*symmetric*] *converse-Un converse-UNION converse-relcomp*
converse-converse converse-Id

1.2 Complete lattices and least fixed points

context *complete-lattice*
begin

1.2.1 A chain-based induction principle

abbreviation *set-chain* :: ′*a set* ⇒ *bool* **where**

set-chain $C \equiv \forall\, x \in C.\ \forall\, y \in C.\ x \leq y \vee y \leq x$

lemma *lfp-chain-induct*:
assumes *mono*: *mono* f
and *step*: $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)$
and *chain*: $\bigwedge C.$ *set-chain* $C \Longrightarrow \forall\ x \in C.\ P\ x \Longrightarrow P\ (Sup\ C)$
shows P (*lfp* f)
unfolding *lfp-eq-fixp*[*OF mono*]
proof (*rule fixp-induct*)
show *monotone* $(\leq)$ $(\leq)$ f **using** *mono* **unfolding** *order-class.mono-def monotone-def* **.**
next
show P (*Sup* $\{\}$) **using** *chain*[*of* $\{\}$] **by** *simp*
next
show *ccpo.admissible Sup* $(\leq)$ P
by (*auto simp add*: *chain ccpo.admissible-def Complete-Partial-Order.chain-def*)
qed *fact*

1.2.2 Preservation of transitivity, asymmetry, irreflexivity by suprema

lemma *trans-Sup-of-chain*:
assumes *set-chain* C **and** *trans*: $\bigwedge R.\ R \in C \Longrightarrow$ *trans* R
shows *trans* (*Sup* C)
proof (*intro transI*)
fix $x\ y\ z$
assume $(x,y) \in Sup\ C$ **and** $(y,z) \in Sup\ C$
from ‹$(x,y) \in Sup\ C$› **obtain** R **where** $R \in C$ **and** $(x,y) \in R$ **by** *blast*
from ‹$(y,z) \in Sup\ C$› **obtain** S **where** $S \in C$ **and** $(y,z) \in S$ **by** *blast*
from ‹$R \in C$› **and** ‹$S \in C$› **and** ‹*set-chain* C› **have** $R \cup S = R \vee R \cup S = S$
by *blast*
with ‹$R \in C$› **and** ‹$S \in C$› **have** $R \cup S \in C$ **by** *fastforce*
with ‹$(x,y) \in R$› **and** ‹$(y,z) \in S$› **and** *trans*[*of* $R \cup S$]
have $(x,z) \in R \cup S$ **unfolding** *trans-def* **by** *blast*
with ‹$R \cup S \in C$› **show** $(x,z) \in \bigcup C$ **by** *blast*
qed

lemma *asym-Sup-of-chain*:
assumes *set-chain* C **and** *asym*: $\bigwedge\ R.\ R \in C \Longrightarrow$ *asym* R
shows *asym* (*Sup* C)
proof (*intro asymI notI*)
fix $a\ b$
assume $(a,b) \in Sup\ C$ **then obtain** R **where** $R \in C$ **and** $(a,b) \in R$ **by** *blast*
assume $(b,a) \in Sup\ C$ **then obtain** S **where** $S \in C$ **and** $(b,a) \in S$ **by** *blast*
from ‹$R \in C$› **and** ‹$S \in C$› **and** ‹*set-chain* C› **have** $R \cup S = R \vee R \cup S = S$
by *blast*
with ‹$R \in C$› **and** ‹$S \in C$› **have** $R \cup S \in C$ **by** *fastforce*
with ‹$(a,b) \in R$› **and** ‹$(b,a) \in S$› **and** *asym*[*THEN asymD*] **show** *False* **by** *blast*
qed

lemma *strict-order-lfp*:
assumes *mono f* **and** $\bigwedge R$. *strict-order R* $\Longrightarrow$ *strict-order* (*f R*)
shows *strict-order* (*lfp f*)
proof (*intro lfp-chain-induct*[*of f strict-order*])
fix *C* :: (′*b* × ′*b*) *set set*
assume *set-chain C* **and** $\forall R \in C$. *strict-order R*
from *this* **show** *strict-order* (*Sup C*)
using *asym-on-iff-irrefl-on-if-trans-on*[*of UNIV*]
by (*metis asym-Sup-of-chain trans-Sup-of-chain*)
qed *fact*+

lemma *trans-lfp*:
assumes *mono f* **and** $\bigwedge R$. *trans R* $\Longrightarrow$ *trans* (*f R*)
shows *trans* (*lfp f*)
by (*metis lfp-chain-induct*[*of f trans*] *assms trans-Sup-of-chain*)

end

1.3 Multiset extension

lemma *mulex-iff-mult*: *mulex r M N* $\longleftrightarrow$ (*M*,*N*) $\in$ *mult* {(*M*,*N*) . *r M N*}
by (*auto simp add*: *mulex-on-def restrict-to-def mult-def mulex1-def tranclp-unfold*)

lemma *multI*:
assumes *trans r M* = *I* + *K N* = *I* + *J J* ≠ {#} $\forall k \in$ *set-mset K*. $\exists j \in$ *set-mset J*. (*k*,*j*) $\in$ *r*
shows (*M*,*N*) $\in$ *mult r*
using *assms one-step-implies-mult* **by** *blast*

lemma *multE*:
assumes *trans r* **and** (*M*,*N*) $\in$ *mult r*
obtains *I J K* **where** *M* = *I* + *K N* = *I* + *J J* ≠ {#} $\forall k \in$ *set-mset K*. $\exists j \in$ *set-mset J*. (*k*,*j*) $\in$ *r*
using *mult-implies-one-step*[*OF assms*] **by** *blast*

lemma *mult-on-union*: (*M*,*N*) $\in$ *mult r* $\Longrightarrow$ (*K* + *M*, *K* + *N*) $\in$ *mult r*
using *mulex-on-union*[*of* $\lambda x\ y$. (*x*,*y*) $\in$ *r UNIV*] **by** (*auto simp*: *mulex-iff-mult*)

lemma *mult-on-union′*: (*M*,*N*) $\in$ *mult r* $\Longrightarrow$ (*M* + *K*, *N* + *K*) $\in$ *mult r*
using *mulex-on-union′*[*of* $\lambda x\ y$. (*x*,*y*) $\in$ *r UNIV*] **by** (*auto simp*: *mulex-iff-mult*)

lemma *mult-on-add-mset*: (*M*,*N*) $\in$ *mult r* $\Longrightarrow$ (*add-mset k M*, *add-mset k N*) $\in$ *mult r*
unfolding *add-mset-add-single*[*of k M*] *add-mset-add-single*[*of k N*] **by** (*rule mult-on-union′*)

lemma *mult-empty*[*simp*]: (*M*,{#}) $\notin$ *mult R*
by (*metis mult-def not-less-empty trancl.cases*)

lemma *mult-singleton*[*simp*]: (*x*, *y*) $\in$ *r* $\Longrightarrow$ (*add-mset x M*, *add-mset y M*) $\in$ *mult*

r
unfolding *add-mset-add-single*[*of x M*] *add-mset-add-single*[*of y M*]
apply (*rule mult-on-union*)
using *mult1-singleton*[*of x y r*] **by** (*auto simp add*: *mult-def mult-on-union*)

lemma *empty-mult*[*simp*]: ({#},*N*) ∈ *mult R* ⟷ *N* ≠ {#}
using *empty-mulex-on*[*of N UNIV* λ*M N*. (*M*,*N*) ∈ *R*] **by** (*auto simp add*: *mulex-iff-mult*)

lemma *trans-mult*: *trans* (*mult R*)
unfolding *mult-def* **by** *simp*

lemma *strict-order-mult*:
assumes *irrefl R* **and** *trans R*
shows *irrefl* (*mult R*) **and** *trans* (*mult R*)
proof −
show *irrefl* (*mult R*) **unfolding** *irrefl-def*
proof (*intro allI notI*, *elim multE*[*OF* ‹*trans R*›])
fix *M I J K*
assume *M* = *I* + *J M* = *I* + *K J* ≠ {#} **and** ∗: ∀ *k* ∈ *set-mset K*. ∃ *j* ∈ *set-mset J*. (*k*, *j*) ∈ *R*
from ‹*M* = *I* + *J*› **and** ‹*M* = *I* + *K*› **have** *J* = *K* **by** *simp*
have *finite* (*set-mset J*) **by** *simp*
then have *set-mset J* = {} **using** ∗ **unfolding** ‹*J* = *K*›
by (*induct rule*: *finite-induct*)
(*simp*, *metis assms insert-absorb insert-iff insert-not-empty irrefl-def transD*)
then show *False* **using** ‹*J* ≠ {#}› **by** *simp*
qed
qed (*simp add*: *trans-mult*)

lemma *mult-of-image-mset*:
assumes *trans R* **and** *trans R′*
and ⋀*x y*. *x* ∈ *set-mset N* ⟹ *y* ∈ *set-mset M* ⟹ (*x*,*y*) ∈ *R* ⟹ (*f x*, *f y*) ∈ *R′*
and (*N*, *M*) ∈ *mult R*
shows (*image-mset f N*, *image-mset f M*) ∈ *mult R′*
proof (*insert assms*(*4*), *elim multE*[*OF assms*(*1*)])
fix *I J K*
assume *N* = *I* + *K M* = *I* + *J J* ≠ {#} ∀ *k* ∈ *set-mset K*. ∃ *j* ∈ *set-mset J*. (*k*, *j*) ∈ *R*
thus (*image-mset f N*, *image-mset f M*) ∈ *mult R′* **using** *assms*(*2*,*3*)
by (*intro multI*) (*auto*, *blast*)
qed

1.4 Incrementality of *mult1* and *mult*

lemma *mono-mult1*: *mono mult1*
unfolding *mono-def mult1-def* **by** *blast*

lemma *mono-mult*: *mono mult*

unfolding *mono-def mult-def*
proof (*intro allI impI subsetI*)
  fix *R S* :: *'a rel* **and** *x*
  assume $R \subseteq S$ **and** $x \in (mult1\ R)^+$
  then show $x \in (mult1\ S)^+$
  using *mono-mult1*[*unfolded mono-def*] *trancl-mono*[*of x mult1 R mult1 S*] **by** *auto*
qed

1.5 Well-orders and well-quasi-orders

lemma *wf-iff-wfp-on*:
  $wf\ p \longleftrightarrow wfp\text{-}on\ (\lambda a\ b.\ (a, b) \in p)\ UNIV$
unfolding *wfp-on-iff-inductive-on wf-def inductive-on-def* **by** *force*

lemma *well-order-implies-wqo*:
  assumes *well-order r*
  shows $wqo\text{-}on\ (\lambda a\ b.\ (a, b) \in r)\ UNIV$
proof (*intro wqo-onI almost-full-onI*)
  show $transp\ (\lambda a\ b.\ (a, b) \in r)$ **using** *assms*
  by (*auto simp only*: *well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def*
    trans-def transp-def)
next
  fix *f* :: $nat \Rightarrow 'a$
  show $good\ (\lambda a\ b.\ (a, b) \in r)\ f$
  using *assms* **unfolding** *well-order-on-def wf-iff-wfp-on wfp-on-def not-ex not-all de-Morgan-conj*
  proof (*elim conjE allE exE*)
    fix *x* **assume** *linear-order r* **and** $f\ x \notin UNIV \vee (f\ (Suc\ x), f\ x) \notin r - Id$
    then have $(f\ x, f\ (Suc\ x)) \in r$ **using** ‹*linear-order r*›
     by (*force simp*: *linear-order-on-def Relation.total-on-def partial-order-on-def preorder-on-def*
      refl-on-def)
    then show $good\ (\lambda a\ b.\ (a, b) \in r)\ f$ **by** (*auto simp*: *good-def*)
  qed
qed

1.6 Splitting lists into prefix, element, and suffix

fun *list-splits* :: $'a\ list \Rightarrow ('a\ list \times 'a \times 'a\ list)\ list$ **where**
  list-splits [] = []
| *list-splits* $(x \# xs) = ([], x, xs) \# map\ (\lambda(xs, x', xs').\ (x \# xs, x', xs'))\ (list\text{-}splits\ xs)$

lemma *list-splits-empty*[*simp*]:
  $list\text{-}splits\ xs = [] \longleftrightarrow xs = []$
by (*cases xs*) *simp-all*

lemma *elem-list-splits-append*:

assumes $(ys,\ y,\ zs) \in set\ (list\text{-}splits\ xs)$
shows $ys\ @\ [y]\ @\ zs = xs$
using *assms* **by** (*induct xs arbitrary*: *ys*) *auto*

lemma *elem-list-splits-length*:
assumes $(ys,\ y,\ zs) \in set\ (list\text{-}splits\ xs)$
shows $length\ ys < length\ xs$ **and** $length\ zs < length\ xs$
using *elem-list-splits-append*[*OF assms*] **by** *auto*

lemma *elem-list-splits-elem*:
assumes $(xs,\ y,\ ys) \in set\ (list\text{-}splits\ zs)$
shows $y \in set\ zs$
using *elem-list-splits-append*[*OF assms*] **by** *force*

lemma *list-splits-append*:
$list\text{-}splits\ (xs\ @\ ys) = map\ (\lambda(xs',\ x',\ ys').\ (xs',\ x',\ ys'\ @\ ys))\ (list\text{-}splits\ xs)\ @$
$map\ (\lambda(xs',\ x',\ ys').\ (xs\ @\ xs',\ x',\ ys'))\ (list\text{-}splits\ ys)$
by (*induct xs*) *auto*

lemma *list-splits-rev*:
$list\text{-}splits\ (rev\ xs) = map\ (\lambda(xs,\ x,\ ys).\ (rev\ ys,\ x,\ rev\ xs))\ (rev\ (list\text{-}splits\ xs))$
by (*induct xs*) (*auto simp add*: *list-splits-append comp-def prod.case-distrib rev-map*)

lemma *list-splits-map*:
$list\text{-}splits\ (map\ f\ xs) = map\ (\lambda(xs,\ x,\ ys).\ (map\ f\ xs,\ f\ x,\ map\ f\ ys))\ (list\text{-}splits\ xs)$
by (*induct xs*) *auto*

end

2 Decreasing Diagrams

theory *Decreasing-Diagrams-II*
imports
Decreasing-Diagrams-II-Aux
HOL−Cardinals.Wellorder-Extension
Abstract−Rewriting.Abstract-Rewriting
begin

2.1 Greek accents

datatype *accent* = *Acute* | *Grave* | *Macron*

lemma *UNIV-accent*: *UNIV* = { *Acute*, *Grave*, *Macron* }
using *accent.nchotomy* **by** *blast*

lemma *finite-accent*: *finite* (*UNIV* :: *accent set*)
by (*simp add*: *UNIV-accent*)

type-synonym *'a letter = accent × 'a*

definition *letter-less* :: *('a × 'a) set ⇒ ('a letter × 'a letter) set* **where**
[simp]: *letter-less R = {(a,b). (snd a, snd b) ∈ R}*

lemma *mono-letter-less*: *mono letter-less*
by (*auto simp add*: *mono-def*)

2.2 Comparing Greek strings

type-synonym *'a greek = 'a letter list*

definition *adj-msog* :: *'a greek ⇒ 'a greek ⇒ ('a letter × 'a greek) ⇒ ('a letter × 'a greek)*
where
adj-msog xs zs l ≡
case l of (y,ys) ⇒ (y, case fst y of Acute ⇒ ys @ zs | Grave ⇒ xs @ ys | Macron ⇒ ys)

definition *ms-of-greek* :: *'a greek ⇒ ('a letter × 'a greek) multiset* **where**
ms-of-greek as = mset
(map (λ(xs, y, zs) ⇒ adj-msog xs zs (y, [])) (list-splits as))

lemma *adj-msog-adj-msog*[*simp*]:
adj-msog xs zs (adj-msog xs' zs' y) = adj-msog (xs @ xs') (zs' @ zs) y
by (*auto simp*: *adj-msog-def split*: *accent.splits prod.splits*)

lemma *compose-adj-msog*[*simp*]: *adj-msog xs zs ∘ adj-msog xs' zs' = adj-msog (xs @ xs') (zs' @ zs)*
by (*simp add*: *comp-def*)

lemma *adj-msog-single*:
adj-msog xs zs (x,[]) = (x, (case fst x of Grave ⇒ xs | Acute ⇒ zs | Macron ⇒ []))
by (*simp add*: *adj-msog-def split*: *accent.splits*)

lemma *ms-of-greek-elem*:
assumes *(x,xs) ∈ set-mset (ms-of-greek ys)*
shows *x ∈ set ys*
using *assms* **by** (*auto dest*: *elem-list-splits-elem simp*: *adj-msog-def ms-of-greek-def*)

lemma *ms-of-greek-shorter*:
assumes *(x, t) ∈# ms-of-greek s*
shows *length s > length t*
using *assms*[*unfolded ms-of-greek-def in-multiset-in-set*]
by (*auto simp*: *elem-list-splits-length adj-msog-def split*: *accent.splits*)

lemma *msog-append*: *ms-of-greek (xs @ ys) = image-mset (adj-msog [] ys) (ms-of-greek xs) +*

image-mset (*adj-msog xs* []) (*ms-of-greek ys*)
by (*auto simp*: *ms-of-greek-def list-splits-append multiset.map-comp comp-def prod.case-distrib*)

definition *nest* :: ('*a* × '*a*) *set* ⇒ ('*a greek* × '*a greek*) *set* ⇒ ('*a greek* × '*a greek*) *set* **where**
[*simp*]: *nest r s* = {(*a*,*b*). (*ms-of-greek a*, *ms-of-greek b*) ∈ *mult* (*letter-less r* <∗*lex*∗> *s*)}

lemma *mono-nest*: *mono* (*nest r*)
unfolding *mono-def*
proof (*intro allI impI subsetI*)
fix *R S x*
assume *1*: *R* ⊆ *S* **and** *2*: *x* ∈ *nest r R*
from *1* **have** *mult* (*letter-less r* <∗*lex*∗> *R*) ⊆ *mult* (*letter-less r* <∗*lex*∗> *S*)
using *mono-mult mono-lex2*[*of letter-less r*] **unfolding** *mono-def* **by** *blast*
with *2* **show** *x* ∈ *nest r S* **by** *auto*
qed

lemma *nest-mono*[*mono-set*]: *x* ⊆ *y* ⟹ (*a*,*b*) ∈ *nest r x* ⟶ (*a*,*b*) ∈ *nest r y*
using *mono-nest*[*unfolded mono-def*, *rule-format*, *of x y r*] **by** *blast*

definition *greek-less* :: ('*a* × '*a*) *set* ⇒ ('*a greek* × '*a greek*) *set* **where**
greek-less r = *lfp* (*nest r*)

lemma *greek-less-unfold*:
greek-less r = *nest r* (*greek-less r*)
using *mono-nest*[*of r*] *lfp-unfold*[*of nest r*] **by** (*simp add*: *greek-less-def*)

2.3 Preservation of strict partial orders

lemma *strict-order-letter-less*:
assumes *strict-order r*
shows *strict-order* (*letter-less r*)
using *assms* **unfolding** *irrefl-def trans-def letter-less-def* **by** *fast*

lemma *strict-order-nest*:
assumes *r*: *strict-order r* **and** *R*: *strict-order R*
shows *strict-order* (*nest r R*)
proof −
have *strict-order* (*mult* (*letter-less r* <∗*lex*∗> *R*))
using *strict-order-letter-less*[*of r*] *irrefl-lex-prod*[*of letter-less r R*]
trans-lex-prod[*of letter-less r R*] *strict-order-mult*[*of letter-less r* <∗*lex*∗> *R*]
assms
by *fast*
from *this* **show** *strict-order* (*nest r R*) **unfolding** *nest-def trans-def irrefl-def*
by *fast*
qed

lemma *strict-order-greek-less*:

assumes *strict-order r*
shows *strict-order* (*greek-less r*)
by (*simp add*: *greek-less-def strict-order-lfp*[*OF mono-nest strict-order-nest*[*OF assms*]])

lemma *trans-letter-less*:
assumes *trans r*
shows *trans* (*letter-less r*)
using *assms* **unfolding** *trans-def letter-less-def* **by** *fast*

lemma *trans-order-nest*: *trans* (*nest r R*)
using *trans-mult* **unfolding** *nest-def trans-def* **by** *fast*

lemma *trans-greek-less*[*simp*]: *trans* (*greek-less r*)
by (*subst greek-less-unfold*) (*rule trans-order-nest*)

lemma *mono-greek-less*: *mono greek-less*
unfolding *greek-less-def mono-def*
proof (*intro allI impI lfp-mono*)
fix *r s* :: ('*a* × '*a*) *set* **and** *R* :: ('*a greek* × '*a greek*) *set*
assume *r* ⊆ *s*
then have *letter-less r* <∗*lex*∗> *R* ⊆ *letter-less s* <∗*lex*∗> *R*
using *mono-letter-less mono-lex1* **unfolding** *mono-def* **by** *metis*
then show *nest r R* ⊆ *nest s R* **using** *mono-mult* **unfolding** *nest-def mono-def*
by *blast*
qed

2.4 Involution

definition *inv-letter* :: '*a letter* ⇒ '*a letter* **where**
inv-letter l ≡
case l of (*a*, *x*) ⇒ (*case a of Grave* ⇒ *Acute* | *Acute* ⇒ *Grave* | *Macron* ⇒ *Macron*, *x*)

lemma *inv-letter-pair*[*simp*]:
inv-letter (*a*, *x*) = (*case a of Grave* ⇒ *Acute* | *Acute* ⇒ *Grave* | *Macron* ⇒ *Macron*, *x*)
by (*simp add*: *inv-letter-def*)

lemma *snd-inv-letter*[*simp*]:
snd (*inv-letter x*) = *snd x*
by (*simp add*: *inv-letter-def split*: *prod.splits*)

lemma *inv-letter-invol*[*simp*]:
inv-letter (*inv-letter x*) = *x*
by (*simp add*: *inv-letter-def split*: *prod.splits accent.splits*)

lemma *inv-letter-mono*[*simp*]:
assumes (*x*, *y*) ∈ *letter-less r*
shows (*inv-letter x*, *inv-letter y*) ∈ *letter-less r*

using *assms* **by** *simp*

definition *inv-greek* :: *'a greek* ⇒ *'a greek* **where**
inv-greek s = *rev* (*map inv-letter s*)

lemma *inv-greek-invol*[*simp*]:
inv-greek (*inv-greek s*) = *s*
by (*simp add*: *inv-greek-def rev-map comp-def*)

lemma *inv-greek-append*:
inv-greek (*s* @ *t*) = *inv-greek t* @ *inv-greek s*
by (*simp add*: *inv-greek-def*)

definition *inv-msog* :: (*'a letter* × *'a greek*) *multiset* ⇒ (*'a letter* × *'a greek*) *multiset* **where**
inv-msog M = *image-mset* (λ(*x*, *t*). (*inv-letter x*, *inv-greek t*)) *M*

lemma *inv-msog-invol*[*simp*]:
inv-msog (*inv-msog M*) = *M*
by (*simp add*: *inv-msog-def multiset.map-comp comp-def prod.case-distrib*)

lemma *ms-of-greek-inv-greek*:
ms-of-greek (*inv-greek M*) = *inv-msog* (*ms-of-greek M*)
unfolding *inv-msog-def inv-greek-def ms-of-greek-def list-splits-rev list-splits-map mset-map*
multiset.map-comp mset-rev inv-letter-def adj-msog-def
by (*rule cong*[*OF cong*[*OF refl*[*of image-mset*]] *refl*]) (*auto split*: *accent.splits*)

lemma *inv-greek-mono*:
assumes *trans r* **and** (*s*, *t*) ∈ *greek-less r*
shows (*inv-greek s*, *inv-greek t*) ∈ *greek-less r*
using *assms*(*2*)
proof (*induct length s* + *length t arbitrary*: *s t rule*: *less-induct*)
note ∗ = *trans-lex-prod*[*OF trans-letter-less*[*OF* ‹*trans r*›] *trans-greek-less*[*of r*]]
case (*less s t*)
have (*inv-msog* (*ms-of-greek s*), *inv-msog* (*ms-of-greek t*)) ∈ *mult* (*letter-less r* <∗*lex*∗> *greek-less r*)
unfolding *inv-msog-def*
proof (*induct rule*: *mult-of-image-mset*[*OF* ∗ ∗])
case (*1 x y*) **thus** *?case*
by (*auto intro*: *less*(*1*) *split*: *prod.splits dest*!: *ms-of-greek-shorter*)
next
case *2* **thus** *?case* **using** *less*(*2*) **by** (*subst*(*asm*) *greek-less-unfold*) *simp*
qed
thus *?case* **by** (*subst greek-less-unfold*) (*auto simp*: *ms-of-greek-inv-greek*)
qed

2.5 Monotonicity of *greek-less r*

lemma *greek-less-rempty*[*simp*]:
(*a*,[]) ∈ *greek-less r* ⟷ *False*
by (*subst greek-less-unfold*) (*auto simp*: *ms-of-greek-def*)

lemma *greek-less-nonempty*:
assumes *b* ≠ []
shows (*a*,*b*) ∈ *greek-less r* ⟷ (*a*,*b*) ∈ *nest r* (*greek-less r*)
by (*subst greek-less-unfold*) *simp*

lemma *greek-less-lempty*[*simp*]:
([],*b*) ∈ *greek-less r* ⟷ *b* ≠ []
proof
assume ([],*b*) ∈ *greek-less r*
then show *b* ≠ [] **using** *greek-less-rempty* **by** *fast*
next
assume *b* ≠ []
then show ([],*b*) ∈ *greek-less r*
unfolding *greek-less-nonempty*[*OF* ‹*b* ≠ []›] **by** (*simp add*: *ms-of-greek-def*)
qed

lemma *greek-less-singleton*:
(*a*, *b*) ∈ *letter-less r* ⟹ ([*a*], [*b*]) ∈ *greek-less r*
by (*subst greek-less-unfold*) (*auto split*: *accent.splits simp*: *adj-msog-def ms-of-greek-def*)

lemma *ms-of-greek-cons*:
ms-of-greek (*x* # *s*) = {# *adj-msog* [] *s* (*x*,[]) #} + *image-mset* (*adj-msog* [*x*] []) (*ms-of-greek s*)
using *msog-append*[*of* [*x*] *s*]
by (*auto simp add*: *adj-msog-def ms-of-greek-def accent.splits*)

lemma *greek-less-cons-mono*:
assumes *trans r*
shows (*s*, *t*) ∈ *greek-less r* ⟹ (*x* # *s*, *x* # *t*) ∈ *greek-less r*
proof (*induct length s* + *length t arbitrary*: *s t rule*: *less-induct*)
note ∗ = *trans-lex-prod*[*OF trans-letter-less*[*OF* ‹*trans r*›] *trans-greek-less*[*of r*]]
case (*less s t*)
{
fix *M* **have** (*M* + *image-mset* (*adj-msog* [*x*] []) (*ms-of-greek s*),
M + *image-mset* (*adj-msog* [*x*] []) (*ms-of-greek t*)) ∈ *mult* (*letter-less r* <∗*lex*∗> *greek-less r*)
proof (*rule mult-on-union*, *induct rule*: *mult-of-image-mset*[*OF* ∗ ∗])
case (*1 x y*) **thus** *?case* **unfolding** *adj-msog-def*
by (*auto intro*: *less*(*1*) *split*: *prod.splits accent.splits dest*!: *ms-of-greek-shorter*)
next
case *2* **thus** *?case* **using** *less*(*2*) **by** (*subst*(*asm*) *greek-less-unfold*) *simp*
qed
}
moreover {

fix N **have** ({# *adj-msog* [] s (x,[]) #} + N,{# *adj-msog* [] t (x,[]) #} + N) ∈
($mult$ (*letter-less* r <∗*lex*∗> *greek-less* r))$^=$
by (*auto simp*: *adj-msog-def less split*: *accent.splits*) **}**
ultimately show *?case* **using** *transD*[*OF trans-mult*]
by (*subst greek-less-unfold*) (*fastforce simp*: *ms-of-greek-cons*)
qed

lemma *greek-less-app-mono2*:
assumes *trans* r **and** (s, t) ∈ *greek-less* r
shows (p @ s, p @ t) ∈ *greek-less* r
using *assms* **by** (*induct* p) (*auto simp add*: *greek-less-cons-mono*)

lemma *greek-less-app-mono1*:
assumes *trans* r **and** (s, t) ∈ *greek-less* r
shows (s @ p, t @ p) ∈ *greek-less* r
using *inv-greek-mono*[*of r inv-greek p* @ *inv-greek s inv-greek p* @ *inv-greek t*]
by (*simp add*: *assms inv-greek-append inv-greek-mono greek-less-app-mono2*)

2.6 Well-founded-ness of *greek-less r*

lemma *greek-embed*:
assumes *trans* r
shows *list-emb* (λa b. (a, b): *reflcl* (*letter-less* r)) a b ⟹ (a, b) ∈ *reflcl* (*greek-less* r)
proof (*induct rule*: *list-emb.induct*)
case (*list-emb-Cons a b y*) **thus** *?case*
using *trans-greek-less*[*unfolded trans-def*] ‹*trans r*›
greek-less-app-mono1[*of r* [] [*y*] *a*] *greek-less-app-mono2*[*of r a b* [*y*]] **by** *auto*
next
case (*list-emb-Cons2 x y a b*) **thus** *?case*
using *trans-greek-less*[*unfolded trans-def*] ‹*trans r*› *greek-less-singleton*[*of x y r*]
greek-less-app-mono1[*of r* [*x*] [*y*] *a*] *greek-less-app-mono2*[*of r a b* [*y*]] **by** *auto*
qed *simp*

lemma *wqo-letter-less*:
assumes t: *trans* r **and** w: *wqo-on* (λa b. (a, b) ∈ $r^=$) *UNIV*
shows *wqo-on* (λa b. (a, b) ∈ (*letter-less* r)$^=$) *UNIV*
proof (*rule wqo-on-hom*[*of id - - prod-le* (=) (λa b. (a, b) ∈ $r^=$), *unfolded image-id id-apply*])
show *wqo-on* (*prod-le* ((=) :: *accent* ⇒ *accent* ⇒ *bool*) (λa b. (a, b) ∈ $r^=$)) *UNIV*
by (*rule dickson*[*OF finite-eq-wqo-on*[*OF finite-accent*] w, *unfolded UNIV-Times-UNIV*])
qed (*insert* t, *auto simp*: *transp-on-def trans-def prod-le-def*)

lemma *wf-greek-less*:
assumes *wf* r **and** *trans* r
shows *wf* (*greek-less* r)
proof −
obtain q **where** r ⊆ q **and** *well-order* q **by** (*metis total-well-order-extension* ‹*wf r*›)

define q' **where** $q' = q - Id$
from ‹*well-order q*› **have** *reflcl* $q' = q$
by (*auto simp add*: *well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def*
refl-on-def q'-def)
from ‹*well-order q*› **have** *trans* q' **and** *irrefl* q'
unfolding *well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def antisym-def*
trans-def irrefl-def q'-def **by** *blast+*
from ‹$r \subseteq q$› ‹*wf r*› **have** $r \subseteq q'$ **by** (*auto simp add*: *q'-def*)
have *wqo-on* ($\lambda a\ b.\ (a,b) \in (\textit{greek-less}\ q')^=$) *UNIV*
proof (*intro wqo-on-hom*[*of id UNIV* ($\lambda a\ b.\ (a,\ b) \in (\textit{greek-less}\ q')^=$)
list-emb ($\lambda a\ b.\ (a,\ b) \in (\textit{letter-less}\ q')^=$), *unfolded surj-id*])
show *transp* ($\lambda a\ b.\ (a,\ b) \in (\textit{greek-less}\ q')^=$)
using *trans-greek-less*[*of q'*] **unfolding** *trans-def transp-on-def* **by** *blast*
next
show $\forall x{\in}\textit{UNIV}.\ \forall y{\in}\textit{UNIV}.\ \textit{list-emb}\ (\lambda a\ b.\ (a,\ b) \in (\textit{letter-less}\ q')^=)\ x\ y \longrightarrow$
$(\textit{id}\ x,\ \textit{id}\ y) \in (\textit{greek-less}\ q')^=$
using *greek-embed*[*OF* ‹*trans q'*›] **by** *auto*
next
show *wqo-on* (*list-emb* ($\lambda a\ b.\ (a,\ b) \in (\textit{letter-less}\ q')^=$)) *UNIV*
using *higman*[*OF wqo-letter-less*[*OF* ‹*trans q'*›]] ‹*well-order q*› ‹*reflcl q' = q*›
by (*auto simp*: *well-order-implies-wqo*)
qed
with *wqo-on-imp-wfp-on*[*OF this*] *strict-order-strict*[*OF strict-order-greek-less*]
‹*irrefl q'*› ‹*trans q'*›
have *wfp-on* ($\lambda a\ b.\ (a,\ b) \in \textit{greek-less}\ q'$) *UNIV* **by** *force*
then show *?thesis*
using *mono-greek-less* ‹$r \subseteq q'$› *wf-subset* **unfolding** *wf-iff-wfp-on*[*symmetric*] *mono-def* **by** *metis*
qed

2.7 Basic Comparisons

lemma *pairwise-imp-mult*:
assumes $N \neq \{\#\}$ **and** $\forall x \in \textit{set-mset}\ M.\ \exists y \in \textit{set-mset}\ N.\ (x,\ y) \in r$
shows $(M,\ N) \in \textit{mult}\ r$
using *assms one-step-implies-mult*[*of - - - {#}*] **by** *auto*

lemma *singleton-greek-less*:
assumes *as*: *snd ‘ set as* $\subseteq$ *under r b*
shows (*as*, [(*a*,*b*)]) $\in$ *greek-less r*
proof −
{
fix *e* **assume** $e \in \textit{set-mset}\ (\textit{ms-of-greek}\ as)$
with *as ms-of-greek-elem*[*of - - as*]
have (*e*, ((*a*,*b*),[])) $\in$ *letter-less r* $<*\textit{lex}*>$ *greek-less r*
by (*cases e*) (*fastforce simp*: *adj-msog-def under-def*)
}

```
  moreover have ms-of-greek [(a,b)] = {# ((a,b),[]) #}
  by (auto simp: ms-of-greek-def adj-msog-def split: accent.splits)
  ultimately show ?thesis
  by (subst greek-less-unfold) (auto intro!: pairwise-imp-mult)
qed

lemma peak-greek-less:
  assumes as: snd ` set as ⊆ under r a and b': b' ∈ {[(Grave,b)],[]}
  and cs: snd ` set cs ⊆ under r a ∪ under r b and a': a' ∈ {[(Acute,a)],[]}
  and bs: snd ` set bs ⊆ under r b
  shows (as @ b' @ cs @ a' @ bs, [(Acute,a),(Grave,b)]) ∈ greek-less r
proof −
  let ?A = (Acute,a) and ?B = (Grave,b)
   have (ms-of-greek (as @ b' @ cs @ a' @ bs), ms-of-greek [?A,?B]) ∈ mult
(letter-less r <*lex*> greek-less r)
  proof (intro pairwise-imp-mult)

    {
      fix e assume e ∈ set-mset (ms-of-greek as)
      with as ms-of-greek-elem[of - - as]
       have (adj-msog [] (b' @ cs @ a' @ bs) e, (?A,[?B])) ∈ letter-less r <*lex*>
greek-less r
      by (cases e) (fastforce simp: adj-msog-def under-def)
    }
    moreover {
      fix e assume e ∈ set-mset (ms-of-greek b')
      with b' singleton-greek-less[OF as] ms-of-greek-elem[of - - b']
        have (adj-msog as (cs @ a' @ bs) e, (?B,[?A])) ∈ letter-less r <*lex*>
greek-less r
      by (cases e) (fastforce simp: adj-msog-def ms-of-greek-def)
    }
    moreover {
      fix e assume e ∈ set-mset (ms-of-greek cs)
      with cs ms-of-greek-elem[of - - cs]
       have (adj-msog (as @ b') (a' @ bs) e, (?A,[?B])) ∈ letter-less r <*lex*>
greek-less r ∨
             (adj-msog (as @ b') (a' @ bs) e, (?B,[?A])) ∈ letter-less r <*lex*>
greek-less r
      by (cases e) (fastforce simp: adj-msog-def under-def)
    }
    moreover {
      fix e assume e ∈ set-mset (ms-of-greek a')
      with a' singleton-greek-less[OF bs] ms-of-greek-elem[of - - a']
        have (adj-msog (as @ b' @ cs) bs e, (?A,[?B])) ∈ letter-less r <*lex*>
greek-less r
      by (cases e) (fastforce simp: adj-msog-def ms-of-greek-def)
    }
    moreover {
      fix e assume e ∈ set-mset (ms-of-greek bs)
```

```
    with bs ms-of-greek-elem[of - - bs]
    have (adj-msog (as @ b' @ cs @ a') [] e, (?B,[?A])) ∈ letter-less r <*lex*>
greek-less r
    by (cases e) (fastforce simp: adj-msog-def under-def)
  }
  moreover have ms-of-greek [?A,?B] = {# (?B,[?A]), (?A,[?B]) #}
  by (simp add: adj-msog-def ms-of-greek-def)
  ultimately show ∀ x∈set-mset (ms-of-greek (as @ b' @ cs @ a' @ bs)).
   ∃ y∈set-mset (ms-of-greek [?A,?B]). (x, y) ∈ letter-less r <*lex*> greek-less r
  by (auto simp: msog-append) blast
 qed (auto simp: ms-of-greek-def)
 then show ?thesis by (subst greek-less-unfold) auto
qed

lemma rcliff-greek-less1:
 assumes trans r
 and as: snd ‘ set as ⊆ under r a ∩ under r b and b': b' ∈ {[(Grave,b)],[]}
 and cs: snd ‘ set cs ⊆ under r b and a': a' = [(Macron,a)]
 and bs: snd ‘ set bs ⊆ under r b
 shows (as @ b' @ cs @ a' @ bs, [(Macron,a),(Grave,b)]) ∈ greek-less r
proof −
 let ?A = (Macron,a) and ?B = (Grave,b)
 have *: ms-of-greek [?A,?B] = {#(?B,[?A]), (?A,[])#} ms-of-greek [?A] = {#(?A,[])#}
 by (simp-all add: ms-of-greek-def adj-msog-def)
 then have **: ms-of-greek [(Macron, a), (Grave, b)] − {#((Macron, a), [])#}
≠ {#}
 by (auto)

 {
  fix e assume e ∈ set-mset (ms-of-greek as)
  with as ms-of-greek-elem[of - - as]
   have (adj-msog [] (b' @ cs @ a' @ bs) e, (?B,[?A])) ∈ letter-less r <*lex*>
greek-less r
  by (cases e) (force simp: adj-msog-def under-def)
 }
 moreover {
  fix e assume e ∈ set-mset (ms-of-greek b')
  with b' singleton-greek-less as ms-of-greek-elem[of - - b']
  have (adj-msog as (cs @ a' @ bs) e, (?B,[?A])) ∈ letter-less r <*lex*> greek-less
r
  by (cases e) (fastforce simp: adj-msog-def ms-of-greek-def)
 }
 moreover {
  fix e assume e ∈ set-mset (ms-of-greek cs)
  with cs ms-of-greek-elem[of - - cs]
   have (adj-msog (as @ b') (a' @ bs) e, (?B,[?A])) ∈ letter-less r <*lex*>
greek-less r
  by (cases e) (fastforce simp: adj-msog-def under-def)
 }
```

```
  moreover {
    fix e assume e ∈ set-mset (ms-of-greek bs)
    with bs ms-of-greek-elem[of - - bs]
     have (adj-msog (as @ b′ @ cs @ a′) [] e, (?B,[?A])) ∈ letter-less r <*lex*>
greek-less r
    by (cases e) (fastforce simp: adj-msog-def under-def)
  }
  moreover have ms-of-greek [?A,?B] = {# (?B,[?A]), (?A,[]) #}
  by (simp add: adj-msog-def ms-of-greek-def)
 ultimately have ∀ x∈set-mset (ms-of-greek (as @ b′ @ cs @ a′ @ bs) − {#(?A,[])#}).
   ∃ y∈set-mset (ms-of-greek [?A,?B] − {#(?A,[])#}). (x, y) ∈ letter-less r <*lex*>
greek-less r
 unfolding msog-append by (auto simp: a′ msog-append ac-simps * adj-msog-single)
  from one-step-implies-mult[OF ** this,of {#(?A,[])#}]
   have (ms-of-greek (as @ b′ @ cs @ a′ @ bs), ms-of-greek [?A,?B]) ∈ mult
(letter-less r <*lex*> greek-less r)
  unfolding a′ msog-append by (auto simp: a′ ac-simps * adj-msog-single)
  then show ?thesis
  by (subst greek-less-unfold) auto
qed

lemma rcliff-greek-less2:
  assumes trans r
  and as: snd ‘ set as ⊆ under r a and b′: b′ ∈ {[(Grave,b)],[]}
  and cs: snd ‘ set cs ⊆ under r a ∪ under r b
  shows (as @ b′ @ cs, [(Macron,a),(Grave,b)]) ∈ greek-less r
proof −
  let ?A = (Macron,a) and ?B = (Grave,b)
   have (ms-of-greek (as @ b′ @ cs), ms-of-greek [?A,?B]) ∈ mult (letter-less r
<*lex*> greek-less r)
  proof (intro pairwise-imp-mult)

    {
      fix e assume e ∈ set-mset (ms-of-greek as)
      with as ms-of-greek-elem[of - - as]
      have (adj-msog [] (b′ @ cs) e, (?A,[])) ∈ letter-less r <*lex*> greek-less r
      by (cases e) (fastforce simp: adj-msog-def under-def)
    }
    moreover {
      fix e assume e ∈ set-mset (ms-of-greek b′)
      with b′ singleton-greek-less[OF as] ms-of-greek-elem[of - - b′]
      have (adj-msog as (cs) e, (?B,[?A])) ∈ letter-less r <*lex*> greek-less r
      by (cases e) (fastforce simp: adj-msog-def ms-of-greek-def)
    }
    moreover {
      fix e assume e ∈ set-mset (ms-of-greek cs)
      with cs ms-of-greek-elem[of - - cs]
      have (adj-msog (as @ b′) [] e, (?A,[])) ∈ letter-less r <*lex*> greek-less r ∨
            (adj-msog (as @ b′) [] e, (?B,[?A])) ∈ letter-less r <*lex*> greek-less r
```

```
    by (cases e) (fastforce simp: adj-msog-def under-def)
  }
  moreover have *: ms-of-greek [?A,?B] = {# (?B,[?A]), (?A,[]) #}
  by (simp add: adj-msog-def ms-of-greek-def)
  ultimately show ∀ x∈set-mset (ms-of-greek (as @ b' @ cs)).
   ∃ y∈set-mset (ms-of-greek [?A,?B]). (x, y) ∈ letter-less r <*lex*> greek-less r
  by (auto simp: msog-append adj-msog-single ac-simps *) blast
 qed (auto simp: ms-of-greek-def)
 then show ?thesis by (subst greek-less-unfold) auto
qed

lemma snd-inv-greek [simp]: snd ` set (inv-greek as) = snd ` set as
by (force simp: inv-greek-def)

lemma lcliff-greek-less1:
 assumes trans r
 and as: snd ` set as ⊆ under r a and b': b' = [(Macron,b)]
 and cs: snd ` set cs ⊆ under r a and a': a' ∈ {[(Acute,a)],[]}
 and bs: snd ` set bs ⊆ under r a ∩ under r b
 shows (as @ b' @ cs @ a' @ bs, [(Acute,a),(Macron,b)]) ∈ greek-less r
proof −
 have *: inv-greek [(Acute,a),(Macron,b)] = [(Macron,b),(Grave,a)] by (simp add:
inv-greek-def)
 have (inv-greek (inv-greek (as @ b' @ cs @ a' @ bs)),
  inv-greek (inv-greek ([(Acute,a),(Macron,b)]))) ∈ greek-less r
  apply (rule inv-greek-mono[OF ‹trans r›])
  apply (unfold inv-greek-append append-assoc *)
  apply (insert assms)
  apply (rule rcliff-greek-less1, auto simp: inv-greek-def)
 done
 then show ?thesis by simp
qed

lemma lcliff-greek-less2:
 assumes trans r
 and cs: snd ` set cs ⊆ under r a ∪ under r b and a': a' ∈ {[(Acute,a)],[]}
 and bs: snd ` set bs ⊆ under r b
 shows (cs @ a' @ bs, [(Acute,a),(Macron,b)]) ∈ greek-less r
proof −
 have *: inv-greek [(Acute,a),(Macron,b)] = [(Macron,b),(Grave,a)] by (simp add:
inv-greek-def)
 have (inv-greek (inv-greek (cs @ a' @ bs)),
   inv-greek (inv-greek ([(Acute,a),(Macron,b)]))) ∈ greek-less r
  apply (rule inv-greek-mono[OF ‹trans r›])
  apply (unfold inv-greek-append append-assoc *)
  apply (insert assms)
  apply (rule rcliff-greek-less2, auto simp: inv-greek-def)
 done
 then show ?thesis by simp
```

qed

2.8 Labeled abstract rewriting

context
fixes *L R E* :: *'b* ⇒ *'a rel*
begin

definition *lstep* :: *'b letter* ⇒ *'a rel* **where**
[*simp*]: *lstep x* = (*case x of* (*a*, *i*) ⇒ (*case a of Acute* ⇒ (*L i*)$^{-1}$ | *Grave* ⇒ *R i*
| *Macron* ⇒ *E i*))

fun *lconv* :: *'b greek* ⇒ *'a rel* **where**
lconv [] = *Id*
| *lconv* (*x* # *xs*) = *lstep x O lconv xs*

lemma *lconv-append*[*simp*]:
lconv (*xs* @ *ys*) = *lconv xs O lconv ys*
by (*induct xs*) *auto*

lemma *conversion-join-or-peak-or-cliff*:
obtains (*join*) *as bs cs* **where** *set as* ⊆ {*Grave*} **and** *set bs* ⊆ {*Macron*} **and**
set cs ⊆ {*Acute*}
and *ds* = *as* @ *bs* @ *cs*
| (*peak*) *as bs* **where** *ds* = *as* @ ([*Acute*] @ [*Grave*]) @ *bs*
| (*lcliff*) *as bs* **where** *ds* = *as* @ ([*Acute*] @ [*Macron*]) @ *bs*
| (*rcliff*) *as bs* **where** *ds* = *as* @ ([*Macron*] @ [*Grave*]) @ *bs*
proof (*induct ds arbitrary*: *thesis*)
case (*Cons d ds thesis*) **note** *IH* = *this* **show** *?case*
proof (*rule IH*(*1*))
fix *as bs* **assume** *ds* = *as* @ ([*Acute*] @ [*Grave*]) @ *bs* **then show** *?case*
using *IH*(*3*)[*of d* # *as bs*] **by** *simp*
next
fix *as bs* **assume** *ds* = *as* @ ([*Acute*] @ [*Macron*]) @ *bs* **then show** *?case*
using *IH*(*4*)[*of d* # *as bs*] **by** *simp*
next
fix *as bs* **assume** *ds* = *as* @ ([*Macron*] @ [*Grave*]) @ *bs* **then show** *?case*
using *IH*(*5*)[*of d* # *as bs*] **by** *simp*
next
fix *as bs cs* **assume** ∗: *set as* ⊆ {*Grave*} *set bs* ⊆ {*Macron*} *set cs* ⊆ {*Acute*}
ds = *as* @ *bs* @ *cs*
show *?case*
proof (*cases d*)
case *Grave* **thus** *?thesis* **using** ∗ *IH*(*2*)[*of d* # *as bs cs*] **by** *simp*
next
case *Macron* **show** *?thesis*
proof (*cases as*)
case *Nil* **thus** *?thesis* **using** ∗ *Macron IH*(*2*)[*of as d* # *bs cs*] **by** *simp*
next

case (*Cons a as*) **thus** *?thesis* **using** ∗ *Macron IH*(*5*)[*of* [] *as* @ *bs* @ *cs*]
by *simp*
qed
next
case *Acute* **show** *?thesis*
proof (*cases as*)
case *Nil* **note** *as* = *this* **show** *?thesis*
proof (*cases bs*)
case *Nil* **thus** *?thesis* **using** ∗ *as Acute IH*(*2*)[*of* [] [] *d* # *cs*] **by** *simp*
next
case (*Cons b bs*) **thus** *?thesis* **using** ∗ *as Acute IH*(*4*)[*of* [] *bs* @ *cs*] **by**
simp
qed
next
case (*Cons a as*) **thus** *?thesis* **using** ∗ *Acute IH*(*3*)[*of* [] *as* @ *bs* @ *cs*] **by**
simp
qed
qed
qed
qed *auto*

lemma *map-eq-append-split*:
assumes *map f xs* = *ys1* @ *ys2*
obtains *xs1 xs2* **where** *ys1* = *map f xs1 ys2* = *map f xs2 xs* = *xs1* @ *xs2*
proof (*insert assms*, *induct ys1 arbitrary*: *xs thesis*)
case (*Cons y ys*) **note** *IH* = *this* **show** *?case*
proof (*cases xs*)
case (*Cons x xs′*) **show** *?thesis*
proof (*rule IH*(*1*))
fix *xs1 xs2* **assume** *ys* = *map f xs1 ys2* = *map f xs2 xs′* = *xs1* @ *xs2* **thus**
?thesis
using *Cons IH*(*2*)[*of x* # *xs1 xs2*] *IH*(*3*) **by** *simp*
next
show *map f xs′* = *ys* @ *ys2* **using** *Cons IH*(*3*) **by** *simp*
qed
qed (*insert Cons*, *simp*)
qed *auto*

lemmas *map-eq-append-splits* = *map-eq-append-split map-eq-append-split*[*OF sym*]

abbreviation *conversion′ M* ≡ ((⋃ *i* ∈ *M*. *R i*) ∪ (⋃ *i* ∈ *M*. *E i*) ∪ (⋃ *i* ∈ *M*. *L*
i)$^{-1}$)*
abbreviation *valley′ M* ≡ (⋃ *i* ∈ *M*. *R i*)* *O* (⋃ *i* ∈ *M*. *E i*)* *O* ((⋃ *i* ∈ *M*. *L*
i)$^{-1}$)*

lemma *conversion-to-lconv*:
assumes (*u*, *v*) ∈ *conversion′ M*
obtains *xs* **where** *snd* ‘ *set xs* ⊆ *M* **and** (*u*, *v*) ∈ *lconv xs*
using *assms*

proof (*induct arbitrary*: *thesis rule*: *converse-rtrancl-induct*)
case *base* **show** *?case* **using** *base*[*of* []] **by** *simp*
next
case (*step u′ x*)
from *step*(*1*) **obtain** *p* **where** *snd p* ∈ *M* **and** (*u′*, *x*) ∈ *lstep p*
by (*force split*: *accent.splits*)
moreover obtain *xs* **where** *snd ‘ set xs* ⊆ *M* (*x*, *v*) ∈ *lconv xs* **by** (*rule step*(*3*))
ultimately show *?case* **using** *step*(*4*)[*of p # xs*] **by** *auto*
qed

definition *lpeak* :: *′b rel* ⇒ *′b* ⇒ *′b* ⇒ *′b greek* ⇒ *bool* **where**
lpeak r a b xs ⟷ (∃ *as b′ cs a′ bs*. *snd ‘ set as* ⊆ *under r a* ∧ *b′* ∈ {[(*Grave*,*b*)],[]} ∧
snd ‘ set cs ⊆ *under r a* ∪ *under r b* ∧ *a′* ∈ {[(*Acute*,*a*)],[]} ∧
snd ‘ set bs ⊆ *under r b* ∧ *xs* = *as* @ *b′* @ *cs* @ *a′* @ *bs*)

definition *lcliff* :: *′b rel* ⇒ *′b* ⇒ *′b* ⇒ *′b greek* ⇒ *bool* **where**
lcliff r a b xs ⟷ (∃ *as b′ cs a′ bs*. *snd ‘ set as* ⊆ *under r a* ∧ *b′* = [(*Macron*,*b*)] ∧
snd ‘ set cs ⊆ *under r a* ∧ *a′* ∈ {[(*Acute*,*a*)],[]} ∧
snd ‘ set bs ⊆ *under r a* ∩ *under r b* ∧ *xs* = *as* @ *b′* @ *cs* @ *a′* @ *bs*) ∨
(∃ *cs a′ bs*. *snd ‘ set cs* ⊆ *under r a* ∪ *under r b* ∧ *a′* ∈ {[(*Acute*,*a*)],[]} ∧
snd ‘ set bs ⊆ *under r b* ∧ *xs* = *cs* @ *a′* @ *bs*)

definition *rcliff* :: *′b rel* ⇒ *′b* ⇒ *′b* ⇒ *′b greek* ⇒ *bool* **where**
rcliff r a b xs ⟷ (∃ *as b′ cs a′ bs*. *snd ‘ set as* ⊆ *under r a* ∩ *under r b* ∧ *b′* ∈ {[(*Grave*,*b*)],[]} ∧
snd ‘ set cs ⊆ *under r b* ∧ *a′* = [(*Macron*,*a*)] ∧
snd ‘ set bs ⊆ *under r b* ∧ *xs* = *as* @ *b′* @ *cs* @ *a′* @ *bs*) ∨
(∃ *as b′ cs*. *snd ‘ set as* ⊆ *under r a* ∧ *b′* ∈ {[(*Grave*,*b*)],[]} ∧
snd ‘ set cs ⊆ *under r a* ∪ *under r b* ∧ *xs* = *as* @ *b′* @ *cs*)

lemma *dd-commute-modulo-conv*[*case-names wf trans peak lcliff rcliff*]:
assumes *wf r* **and** *trans r*
and *pk*: ⋀*a b s t u*. (*s*, *t*) ∈ *L a* ⟹ (*s*, *u*) ∈ *R b* ⟹ ∃ *xs*. *lpeak r a b xs* ∧ (*t*, *u*) ∈ *lconv xs*
and *lc*: ⋀*a b s t u*. (*s*, *t*) ∈ *L a* ⟹ (*s*, *u*) ∈ *E b* ⟹ ∃ *xs*. *lcliff r a b xs* ∧ (*t*, *u*) ∈ *lconv xs*
and *rc*: ⋀*a b s t u*. (*s*, *t*) ∈ (*E a*)$^{-1}$ ⟹ (*s*, *u*) ∈ *R b* ⟹ ∃ *xs*. *rcliff r a b xs* ∧ (*t*, *u*) ∈ *lconv xs*
shows *conversion′ UNIV* ⊆ *valley′ UNIV*
proof (*intro subrelI*)
fix *u v*
assume (*u*,*v*) ∈ *conversion′ UNIV*
then obtain *xs* **where** (*u*, *v*) ∈ *lconv xs* **by** (*auto intro*: *conversion-to-lconv*[*of u v*])
then show (*u*, *v*) ∈ *valley′ UNIV*
proof (*induct xs rule*: *wf-induct*[*of greek-less r*])
case *1* **thus** *?case* **using** *wf-greek-less*[*OF* ‹*wf r*› ‹*trans r*›] **.**

```
  next
    case (2 xs) show ?case
    proof (rule conversion-join-or-peak-or-cliff[of map fst xs])
      fix as bs cs
      assume ∗: set as ⊆ {Grave} set bs ⊆ {Macron} set cs ⊆ {Acute} map fst xs
= as @ bs @ cs
      then show (u, v) ∈ valley′ UNIV
      proof (elim map-eq-append-splits)
        fix as′ bs′ cs′ bcs′
        assume as: set as ⊆ {Grave} as = map fst as′ and
          bs: set bs ⊆ {Macron} bs = map fst bs′ and
          cs: set cs ⊆ {Acute} cs = map fst cs′ and
          xs: xs = as′ @ bcs′ bcs′ = bs′ @ cs′
        from as(1)[unfolded as(2)] have as′: ⋀x y. (x,y) ∈ lconv as′ ⟹ (x,y) ∈
(⋃a. R a)∗
        proof (induct as′)
          case (Cons x′ xs)
          have ⋀x y z i. (x,y) ∈ R i ⟹ (y,z) ∈ (⋃a. R a)∗ ⟹ (x,z) ∈ (⋃a. R
a)∗
            by (rule rtrancl-trans) auto
          with Cons show ?case by auto
        qed simp
        from bs(1)[unfolded bs(2)] have bs′: ⋀x y. (x,y) ∈ lconv bs′ ⟹ (x,y) ∈
(⋃a. E a)∗
        proof (induct bs′)
          case (Cons x′ xs)
          have ⋀x y z i. (x,y) ∈ E i ⟹ (y,z) ∈ (⋃a. E a)∗ ⟹ (x,z) ∈ (⋃a. E a)∗
            by (rule rtrancl-trans) auto
          with Cons show ?case by auto
        qed simp
        from cs(1)[unfolded cs(2)] have cs′: ⋀x y. (x,y) ∈ lconv cs′ ⟹ (x,y) ∈
((⋃a. L a)−1)∗
        proof (induct cs′)
          case (Cons x′ xs)
          have ⋀x y z i. (x,y) ∈ (L i)−1 ⟹ (y,z) ∈ ((⋃a. L a)−1)∗ ⟹ (x,z) ∈
((⋃a. L a)−1)∗
            by (rule rtrancl-trans) auto
          with Cons show ?case by auto
        qed simp
        from 2(2) as′ bs′ cs′ show (u, v) ∈ valley′ UNIV
        unfolding xs lconv-append by auto (meson relcomp.simps)
      qed
    next
      fix as bs assume ∗: map fst xs = as @ ([Acute] @ [Grave]) @ bs
      {
        fix p a b q t′ s′ u′
        assume xs: xs = p @ [(Acute,a),(Grave,b)] @ q and p: (u,t′) ∈ lconv p
          and a: (s′,t′) ∈ L a and b: (s′,u′) ∈ R b and q: (u′,v) ∈ lconv q
        obtain js where lp: lpeak r a b js and js: (t′,u′) ∈ lconv js using pk[OF a
```

```
b] by auto
      from lp have (js, [(Acute,a),(Grave,b)]) ∈ greek-less r
      unfolding lpeak-def using peak-greek-less[of - r a - b] by fastforce
      then have (p @ js @ q, xs) ∈ greek-less r unfolding xs
      by (intro greek-less-app-mono1 greek-less-app-mono2 ‹trans r›) auto
      moreover have (u, v) ∈ lconv (p @ js @ q)
      using p q js by auto
      ultimately have (u, v) ∈ valley′ UNIV using 2(1) by blast
    }
    with ∗ show (u, v) ∈ valley′ UNIV using 2(2)
     by (auto elim!: map-eq-append-splits relcompEpair simp del: append.simps)
simp
  next
    fix as bs assume ∗: map fst xs = as @ ([Acute] @ [Macron]) @ bs
    {
      fix p a b q t′ s′ u′
      assume xs: xs = p @ [(Acute,a),(Macron,b)] @ q and p: (u,t′) ∈ lconv p
        and a: (s′,t′) ∈ L a and b: (s′,u′) ∈ E b and q: (u′,v) ∈ lconv q
      obtain js where lp: lcliff r a b js and js: (t′,u′) ∈ lconv js using lc[OF a
b] by auto
      from lp have (js, [(Acute,a),(Macron,b)]) ∈ greek-less r
      unfolding lcliff-def
       using lcliff-greek-less1[OF ‹trans r›, of - a - b] lcliff-greek-less2[OF ‹trans
r›, of - a b]
      by fastforce
      then have (p @ js @ q, xs) ∈ greek-less r unfolding xs
      by (intro greek-less-app-mono1 greek-less-app-mono2 ‹trans r›) auto
      moreover have (u, v) ∈ lconv (p @ js @ q)
      using p q js by auto
      ultimately have (u, v) ∈ valley′ UNIV using 2(1) by blast
    }
    with ∗ show (u, v) ∈ valley′ UNIV using 2(2)
     by (auto elim!: map-eq-append-splits relcompEpair simp del: append.simps)
simp
  next
    fix as bs assume ∗: map fst xs = as @ ([Macron] @ [Grave]) @ bs
    {
      fix p a b q t′ s′ u′
      assume xs: xs = p @ [(Macron,a),(Grave,b)] @ q and p: (u,t′) ∈ lconv p
        and a: (s′,t′) ∈ (E a)⁻¹ and b: (s′,u′) ∈ R b and q: (u′,v) ∈ lconv q
      obtain js where lp: rcliff r a b js and js: (t′,u′) ∈ lconv js using rc[OF a
b] by auto
      from lp have (js, [(Macron,a),(Grave,b)]) ∈ greek-less r
      unfolding rcliff-def
       using rcliff-greek-less1[OF ‹trans r›, of - a b] rcliff-greek-less2[OF ‹trans
r›, of - a - b]
      by fastforce
      then have (p @ js @ q, xs) ∈ greek-less r unfolding xs
      by (intro greek-less-app-mono1 greek-less-app-mono2 ‹trans r›) auto
```

moreover have $(u, v) \in lconv\ (p\ @\ js\ @\ q)$
using *p q js* **by** *auto*
ultimately have $(u, v) \in valley'\ UNIV$ **using** *2(1)* **by** *blast*
}
with $*$ **show** $(u, v) \in valley'\ UNIV$ **using** *2(2)*
by (*auto elim*!: *map-eq-append-splits relcompEpair simp del*: *append.simps*) *simp*
qed
qed
qed

3 Results

3.1 Church-Rosser modulo

Decreasing diagrams for Church-Rosser modulo, commutation version.

lemma *dd-commute-modulo*[*case-names wf trans peak lcliff rcliff*]:
assumes *wf r* **and** *trans r*
and *pk*: $\bigwedge a\ b\ s\ t\ u.\ (s, t) \in L\ a \Longrightarrow (s, u) \in R\ b \Longrightarrow$
$(t, u) \in conversion'\ (under\ r\ a)\ O\ (R\ b)^{=}\ O\ conversion'\ (under\ r\ a \cup under\ r\ b)\ O$
$((L\ a)^{-1})^{=}\ O\ conversion'\ (under\ r\ b)$
and *lc*: $\bigwedge a\ b\ s\ t\ u.\ (s, t) \in L\ a \Longrightarrow (s, u) \in E\ b \Longrightarrow$
$(t, u) \in conversion'\ (under\ r\ a)\ O\ E\ b\ O\ conversion'\ (under\ r\ a)\ O$
$((L\ a)^{-1})^{=}\ O\ conversion'\ (under\ r\ a \cap under\ r\ b) \vee$
$(t, u) \in conversion'\ (under\ r\ a \cup under\ r\ b)\ O\ ((L\ a\ )^{-1})^{=}\ O\ conversion'\ (under\ r\ b)$
and *rc*: $\bigwedge a\ b\ s\ t\ u.\ (s, t) \in (E\ a)^{-1} \Longrightarrow (s, u) \in R\ b \Longrightarrow$
$(t, u) \in conversion'\ (under\ r\ a \cap under\ r\ b)\ O\ (R\ b)^{=}\ O\ conversion'\ (under\ r\ b)\ O$
$E\ a\ O\ conversion'\ (under\ r\ b) \vee$
$(t, u) \in conversion'\ (under\ r\ a)\ O\ (R\ b)^{=}\ O\ conversion'\ (under\ r\ a \cup under\ r\ b)$
shows $conversion'\ UNIV \subseteq valley'\ UNIV$
proof (*cases rule*: *dd-commute-modulo-conv*[*of r*])
case (*peak a b s t u*)
{
fix *w x y z*
assume $(t, w) \in conversion'\ (under\ r\ a)$
from *conversion-to-lconv*[*OF this*]
obtain *as* **where** *snd ' set as* $\subseteq$ $under\ r\ a$ $(t, w) \in lconv\ as$ **by** *auto*
moreover assume $(w, x) \in (R\ b)^{=}$
then obtain b' **where** $b' \in \{[(Grave,b)],[]\}$ $(w, x) \in lconv\ b'$ **by** *fastforce*
moreover assume $(x, y) \in conversion'\ (under\ r\ a \cup under\ r\ b)$
from *conversion-to-lconv*[*OF this*]
obtain *cs* **where** *snd ' set cs* $\subseteq$ $under\ r\ a \cup under\ r\ b$ $(x, y) \in lconv\ cs$ **by** *auto*
moreover assume $(y, z) \in ((L\ a)^{-1})^{=}$

```
    then obtain a' where a' ∈ {[(Acute,a)],[]} (y, z) ∈ lconv a' by fastforce
    moreover assume (z, u) ∈ conversion' (under r b)
    from conversion-to-lconv[OF this]
    obtain bs where snd ‘ set bs ⊆ under r b (z, u) ∈ lconv bs by auto
    ultimately have ∃xs. lpeak r a b xs ∧ (t, u) ∈ lconv xs
    by (intro exI[of - as @ b' @ cs @ a' @ bs], unfold lconv-append lpeak-def) blast
  }
  then show ?case using pk[OF peak] by blast
next
  case (lcliff a b s t u)
  {
    fix w x y z
    assume (t, w) ∈ conversion' (under r a)
    from conversion-to-lconv[OF this]
    obtain as where snd ‘ set as ⊆ under r a (t, w) ∈ lconv as by auto
    moreover assume (w, x) ∈ E b
    then obtain b' where b' = [(Macron,b)] (w, x) ∈ lconv b' by fastforce
    moreover assume (x, y) ∈ conversion' (under r a)
    from conversion-to-lconv[OF this]
    obtain cs where snd ‘ set cs ⊆ under r a (x, y) ∈ lconv cs by auto
    moreover assume (y, z) ∈ ((L a)^-1)^=
    then obtain a' where a' ∈ {[(Acute,a)],[]} (y, z) ∈ lconv a' by fastforce
    moreover assume (z, u) ∈ conversion' (under r a ∩ under r b)
    from conversion-to-lconv[OF this]
    obtain bs where snd ‘ set bs ⊆ under r a ∩ under r b (z, u) ∈ lconv bs by
auto
    ultimately have ∃xs. lcliff r a b xs ∧ (t, u) ∈ lconv xs
    by (intro exI[of - as @ b' @ cs @ a' @ bs], unfold lconv-append lcliff-def) blast
  }
  moreover {
    fix w x
    assume (t, w) ∈ conversion' (under r a ∪ under r b)
    from conversion-to-lconv[OF this]
    obtain cs where snd ‘ set cs ⊆ under r a ∪ under r b (t, w) ∈ lconv cs by
auto
    moreover assume (w, x) ∈ ((L a)^-1)^=
    then obtain a' where a' ∈ {[(Acute,a)],[]} (w, x) ∈ lconv a' by fastforce
    moreover assume (x, u) ∈ conversion' (under r b)
    from conversion-to-lconv[OF this]
    obtain bs where snd ‘ set bs ⊆ under r b (x, u) ∈ lconv bs by auto
    ultimately have ∃xs. lcliff r a b xs ∧ (t, u) ∈ lconv xs
    by (intro exI[of - cs @ a' @ bs], unfold lconv-append lcliff-def) blast
  }
  ultimately show ?case using lc[OF lcliff] by blast
next
  case (rcliff a b s t u)
  {
    fix w x y z
    assume (t, w) ∈ conversion' (under r a ∩ under r b)
```

from *conversion-to-lconv*[*OF this*]
obtain *as* **where** *snd ‘ set as* ⊆ *under r a* ∩ *under r b* (*t*, *w*) ∈ *lconv as* **by** *auto*
moreover assume (*w*, *x*) ∈ (*R b*)$^{=}$
then obtain *b′* **where** *b′* ∈ {[(*Grave*,*b*)],[]} (*w*, *x*) ∈ *lconv b′* **by** *fastforce*
moreover assume (*x*, *y*) ∈ *conversion′* (*under r b*)
from *conversion-to-lconv*[*OF this*]
obtain *cs* **where** *snd ‘ set cs* ⊆ *under r b* (*x*, *y*) ∈ *lconv cs* **by** *auto*
moreover assume (*y*, *z*) ∈ *E a*
then obtain *a′* **where** *a′* = [(*Macron*,*a*)] (*y*, *z*) ∈ *lconv a′* **by** *fastforce*
moreover assume (*z*, *u*) ∈ *conversion′* (*under r b*)
from *conversion-to-lconv*[*OF this*]
obtain *bs* **where** *snd ‘ set bs* ⊆ *under r b* (*z*, *u*) ∈ *lconv bs* **by** *auto*
ultimately have ∃ *xs*. *rcliff r a b xs* ∧ (*t*, *u*) ∈ *lconv xs*
by (*intro exI*[*of - as* @ *b′* @ *cs* @ *a′* @ *bs*], *unfold lconv-append rcliff-def*) *blast*
}
moreover {
fix *w x*
assume (*t*, *w*) ∈ *conversion′* (*under r a*)
from *conversion-to-lconv*[*OF this*]
obtain *as* **where** *snd ‘ set as* ⊆ *under r a* (*t*, *w*) ∈ *lconv as* **by** *auto*
moreover assume (*w*, *x*) ∈ (*R b*)$^{=}$
then obtain *b′* **where** *b′* ∈ {[(*Grave*,*b*)],[]} (*w*, *x*) ∈ *lconv b′* **by** *fastforce*
moreover assume (*x*, *u*) ∈ *conversion′* (*under r a* ∪ *under r b*)
from *conversion-to-lconv*[*OF this*]
obtain *cs* **where** *snd ‘ set cs* ⊆ *under r a* ∪ *under r b* (*x*, *u*) ∈ *lconv cs* **by** *auto*
ultimately have ∃ *xs*. *rcliff r a b xs* ∧ (*t*, *u*) ∈ *lconv xs*
by (*intro exI*[*of - as* @ *b′* @ *cs*], *unfold lconv-append rcliff-def*) *blast*
}
ultimately show *?case* **using** *rc*[*OF rcliff*] **by** *blast*
qed *fact+*

end

Decreasing diagrams for Church-Rosser modulo.

lemma *dd-cr-modulo*[*case-names wf trans symE peak cliff*]:
assumes *wf r* **and** *trans r* **and** *E*: ⋀*i*. *sym* (*E i*)
and *pk*: ⋀*a b s t u*. (*s*, *t*) ∈ *L a* ⟹ (*s*, *u*) ∈ *L b* ⟹
(*t*, *u*) ∈ *conversion′ L L E* (*under r a*) *O* (*L b*)$^{=}$ *O conversion′ L L E* (*under r a* ∪ *under r b*) *O*
((*L a*)$^{-1}$)$^{=}$ *O conversion′ L L E* (*under r b*)
and *cl*: ⋀*a b s t u*. (*s*, *t*) ∈ *L a* ⟹ (*s*, *u*) ∈ *E b* ⟹
(*t*, *u*) ∈ *conversion′ L L E* (*under r a*) *O E b O conversion′ L L E* (*under r a*) *O*
((*L a*)$^{-1}$)$^{=}$ *O conversion′ L L E* (*under r a* ∩ *under r b*) ∨
(*t*, *u*) ∈ *conversion′ L L E* (*under r a* ∪ *under r b*) *O* ((*L a*)$^{-1}$)$^{=}$ *O conversion′ L L E* (*under r b*)
shows *conversion′ L L E UNIV* ⊆ *valley′ L L E UNIV*

proof (*induct rule*: *dd-commute-modulo*[*of r*])
note $E' = E$[*unfolded sym-conv-converse-eq*]
case (*rcliff a b s t u*) **show** *?case*
using *cl*[*OF rcliff*(*2*) *rcliff*(*1*)[*unfolded E'*], *unfolded converse-iff*[*of t u*,*symmetric*]]
by (*auto simp only*: E' *converse-inward*) (*auto simp only*: *ac-simps*)
qed *fact+*

3.2 Commutation and confluence

abbreviation *conversion''* $L\ R\ M \equiv ((\bigcup i \in M.\ R\ i) \cup (\bigcup i \in M.\ L\ i)^{-1})^*$
abbreviation *valley''* $L\ R\ M \equiv (\bigcup i \in M.\ R\ i)^*\ O\ ((\bigcup i \in M.\ L\ i)^{-1})^*$

Decreasing diagrams for commutation.

lemma *dd-commute*[*case-names wf trans peak*]:
assumes *wf r* **and** *trans r*
and *pk*: $\bigwedge a\ b\ s\ t\ u.\ (s, t) \in L\ a \Longrightarrow (s, u) \in R\ b \Longrightarrow$
$(t, u) \in$ *conversion''* $L\ R$ (*under r a*) O $(R\ b)^=$ O *conversion''* $L\ R$ (*under r a* $\cup$ *under r b*) O
$((L\ a)^{-1})^=$ O *conversion''* $L\ R$ (*under r b*)
shows *commute* $(\bigcup i.\ L\ i)$ $(\bigcup i.\ R\ i)$
proof −
have $((\bigcup i.\ L\ i)^{-1})^*$ O $(\bigcup i.\ R\ i)^* \subseteq$ *conversion''* $L\ R$ *UNIV* **by** *regexp*
also have $\ldots \subseteq$ *valley''* $L\ R$ *UNIV*
using *dd-commute-modulo*[*OF assms*(*1*,*2*), *of L R* λ-. {}] *pk* **by** *auto*
finally show *?thesis* **by** (*simp only*: *commute-def*)
qed

Decreasing diagrams for confluence.

lemmas *dd-cr*[*case-names wf trans peak*] =
dd-commute[*of - L L* **for** *L*, *unfolded CR-iff-self-commute*[*symmetric*]]

3.3 Extended decreasing diagrams

context
fixes *r q* :: *'b rel*
assumes *wf r* **and** *trans r* **and** *trans q* **and** *refl q* **and** *compat*: $r\ O\ q \subseteq r$
begin

private abbreviation (*input*) *down* :: $('b \Rightarrow 'a\ rel) \Rightarrow ('b \Rightarrow 'a\ rel)$ **where**
down $L \equiv \lambda i.\ \bigcup j \in$ *under q i*. $L\ j$

private lemma *Union-down*: $(\bigcup i.\ down\ L\ i) = (\bigcup i.\ L\ i)$
using ‹*refl q*› **by** (*auto simp*: *refl-on-def under-def*)

Extended decreasing diagrams for commutation.

lemma *edd-commute*[*case-names wf transr transq reflq compat peak*]:
assumes *pk*: $\bigwedge a\ b\ s\ t\ u.\ (s, t) \in L\ a \Longrightarrow (s, u) \in R\ b \Longrightarrow$
$(t, u) \in$ *conversion''* $L\ R$ (*under r a*) O (*down R b*)$^=$ O *conversion''* $L\ R$ (*under r a* $\cup$ *under r b*) O

```
    ((down L a)^-1)^= O conversion'' L R (under r b)
  shows commute (⋃ i. L i) (⋃ i. R i)
unfolding Union-down[of L, symmetric] Union-down[of R, symmetric]
proof (induct rule: dd-commute[of r down L down R])
  case (peak a b s t u)
  then obtain a' b' where a': (a', a) ∈ q (s, t) ∈ L a' and b': (b', b) ∈ q (s, u)
∈ R b'
  by (auto simp: under-def)
  have ⋀a' a. (a',a) ∈ q ⟹ under r a' ⊆ under r a using compat by (auto simp:
under-def)
  then have aux1: ⋀a' a L. (a',a) ∈ q ⟹ (⋃ i ∈ under r a'. L i) ⊆ (⋃ i ∈ under
r a. L i) by auto
  have aux2: ⋀a' a L. (a',a) ∈ q ⟹ down L a' ⊆ down L a
    using ‹trans q› by (auto simp: under-def trans-def)
  have aux3: ⋀a L. (⋃ i ∈ under r a. L i) ⊆ (⋃ i ∈ under r a. down L i)
    using ‹refl q› by (auto simp: under-def refl-on-def)
  from aux1[OF a'(1), of L] aux1[OF a'(1), of R] aux2[OF a'(1), of L]
      aux1[OF b'(1), of L] aux1[OF b'(1), of R] aux2[OF b'(1), of R]
      aux3[of L] aux3[of R]
  show ?case
  by (intro subsetD[OF - pk[OF ‹(s, t) ∈ L a'› ‹(s, u) ∈ R b'›]], unfold UN-Un)
     (intro relcomp-mono rtrancl-mono Un-mono iffD2[OF converse-mono]; fast)
qed fact+
```

Extended decreasing diagrams for confluence.

```
lemmas edd-cr[case-names wf transr transq reflq compat peak] =
  edd-commute[of L L for L, unfolded CR-iff-self-commute[symmetric]]

end

end
```

References

[1] B. Felgenhauer and V. van Oostrom. Proof orders for decreasing diagrams. In *Proc. 24th International Conference on Rewriting Techniques and Applications*, number 21 in Leibniz International Proceedings in Informatics, pages 174–189, 2013.

[2] N. Hirokawa and A. Middeldorp. Decreasing diagrams and relative termination. In *Proc. 5th International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Artificial Intelligence*, pages 487–501, 2010.

[3] V. van Oostrom. Confluence by decreasing diagrams – converted. In *Proc. 19th International Conference on Rewriting Techniques and Ap-*

plications, volume 5117 of *Lecture Notes in Computer Science*, pages 306–320, 2008.