

Decreasing-Diagrams-II

By Bertram Felgenhauer

December 14, 2021

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	3
1.2.1	A chain-based induction principle	3
1.2.2	Preservation of transitivity, asymmetry, irreflexivity by suprema	3
1.3	Multiset extension	4
1.4	Incrementality of <i>mult1</i> and <i>mult</i>	6
1.5	Well-orders and well-quasi-orders	6
1.6	Splitting lists into prefix, element, and suffix	7
2	Decreasing Diagrams	7
2.1	Greek accents	8
2.2	Comparing Greek strings	8
2.3	Preservation of strict partial orders	9

2.4	Involution	10
2.5	Monotonicity of <i>greek-less</i> r	12
2.6	Well-founded-ness of <i>greek-less</i> 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

lemma *asymI2*: $(\bigwedge a b. (a, b) \in R \implies (b, a) \notin R) \implies \text{asym } R$
by (*metis asymI irrefl-def*)

abbreviation *strict-order* $R \equiv \text{irrefl } R \wedge \text{trans } R$

lemma *order-asym*: $\text{trans } R \implies \text{asym } R = \text{irrefl } R$

unfolding *asym.simps irrefl-def trans-def* **by** *meson*

lemma *strict-order-strict*: $\text{strict-order } q \implies \text{strict } (\lambda a b. (a, b) \in q^\equiv) = (\lambda a b. (a, b) \in q)$

unfolding *trans-def irrefl-def* **by** *fast*

lemma *mono-lex1*: $\text{mono } (\lambda r. \text{lex-prod } r \ s)$

by (*auto simp add: mono-def*)

lemma *mono-lex2*: $\text{mono } (\text{lex-prod } r)$

by (*auto simp add: mono-def*)

lemma *irrefl-lex-prod*: $\text{irrefl } R \implies \text{irrefl } S \implies \text{irrefl } (R \langle * \text{lex} * \rangle S)$

by (*auto simp add: lex-prod-def irrefl-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 \Rightarrow 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*: $\bigwedge C. \text{set-chain } C \Longrightarrow \forall x \in C. P\ x \Longrightarrow P\ (\text{Sup } C)$

shows *P* (*lfp* *f*)

unfolding *lfp-eq-fixp*[*OF* *mono*]

proof (*rule* *fixp-induct*)

show *monotone* (\leq) (\leq) *f* **using** *mono* **unfolding** *order-class.monotone-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*: $\bigwedge R. R \in C \Longrightarrow \text{trans } R$

shows *trans* (*Sup* *C*)

proof (*intro* *transI*)

fix *x* *y* *z*

assume $\langle x, y \rangle \in \text{Sup } C$ **and** $\langle y, z \rangle \in \text{Sup } C$

from $\langle x, y \rangle \in \text{Sup } C$ **obtain** *R* **where** $R \in C$ **and** $\langle x, y \rangle \in R$ **by** *blast*

from $\langle y, z \rangle \in \text{Sup } C$ **obtain** *S* **where** $S \in C$ **and** $\langle y, z \rangle \in S$ **by** *blast*

from $\langle R \in C \rangle$ **and** $\langle S \in C \rangle$ **and** $\langle \text{set-chain } C \rangle$ **have** $R \cup S = R \vee R \cup S = S$

by *blast*

with $\langle R \in C \rangle$ **and** $\langle S \in C \rangle$ **have** $R \cup S \in C$ **by** *fastforce*

with $\langle x, y \rangle \in R$ **and** $\langle y, z \rangle \in S$ **and** *trans*[*of* $R \cup S$]

have $\langle x, z \rangle \in R \cup S$ **unfolding** *trans-def* **by** *blast*

with $\langle R \cup S \in C \rangle$ **show** $\langle x, z \rangle \in \bigcup C$ **by** *blast*

qed

lemma *asym-Sup-of-chain*:

assumes *set-chain* *C* **and** *asym*: $\bigwedge R. R \in C \Longrightarrow \text{asym } R$

shows *asym* (*Sup C*)
proof (*intro asymI2 notI*)
fix *a b*
assume $(a,b) \in \text{Sup } C$ **then obtain** *R* **where** $R \in C$ **and** $(a,b) \in R$ **by** *blast*
assume $(b,a) \in \text{Sup } C$ **then obtain** *S* **where** $S \in C$ **and** $(b,a) \in S$ **by** *blast*
from $\langle R \in C \rangle$ **and** $\langle S \in C \rangle$ **and** $\langle \text{set-chain } C \rangle$ **have** $R \cup S = R \vee R \cup S = S$
by *blast*
with $\langle R \in C \rangle$ **and** $\langle S \in C \rangle$ **have** $R \cup S \in C$ **by** *fastforce*
with $\langle (a,b) \in R \rangle$ **and** $\langle (b,a) \in S \rangle$ **and** *asym* **show** *False* **unfolding** *asym.simps*
by *blast*
qed

lemma *strict-order-lfp*:
assumes *mono f* **and** $\bigwedge R. \text{strict-order } R \implies \text{strict-order } (f R)$
shows *strict-order* (*lfp f*)
proof (*intro lfp-chain-induct[of f strict-order]*)
fix *C* :: $'b \times 'b$ *set set*
assume *set-chain C* **and** $\forall R \in C. \text{strict-order } R$
from *this* **show** *strict-order* (*Sup C*) **by** (*metis asym-Sup-of-chain trans-Sup-of-chain order-asym*)
qed *fact+*

lemma *trans-lfp*:
assumes *mono f* **and** $\bigwedge R. \text{trans } R \implies \text{trans } (f R)$
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*: $\text{mulex } r \ M \ N \longleftrightarrow (M,N) \in \text{mult } \{(M,N) . r \ M \ N\}$
by (*auto simp add: mulex-on-def restrict-to-def mult-def mulex1-def tranclp-unfold*)

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

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

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

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

lemma *mult-on-add-mset*: $(M, N) \in \text{mult } r \implies (\text{add-mset } k \ M, \text{add-mset } k \ N) \in \text{mult } r$
unfolding *add-mset-add-single*[of $k \ M$] *add-mset-add-single*[of $k \ N$] **by** (*rule mult-on-union'*)

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

lemma *mult-singleton[simp]*: $(x, y) \in r \implies (\text{add-mset } x \ M, \text{add-mset } y \ M) \in \text{mult } r$
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]*: $(\{\#\}, N) \in \text{mult } R \iff N \neq \{\#\}$
using *empty-mulex-on*[of $N \text{ UNIV } \lambda M N. (M, N) \in R$] **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 $M = I + J \ M = I + K \ J \neq \{\#\}$ **and** $*$: $\forall k \in \text{set-mset } K. \exists j \in \text{set-mset } J. (k, j) \in R$
from $\langle M = I + J \rangle$ **and** $\langle M = I + K \rangle$ **have** $J = K$ **by** *simp*
have *finite (set-mset J)* **by** *simp*
then have $\text{set-mset } J = \{\}$ **using** $*$ **unfolding** $\langle J = K \rangle$
by (*induct rule: finite-induct*)
(simp, metis assms insert-absorb insert-iff insert-not-empty irrefl-def transD)
then show *False* **using** $\langle J \neq \{\#\} \rangle$ **by** *simp*
qed
qed (*simp add: trans-mult*)

lemma *mult-of-image-mset*:
assumes *trans R and trans R'*
and $\bigwedge x y. x \in \text{set-mset } N \implies y \in \text{set-mset } M \implies (x, y) \in R \implies (f \ x, f \ y) \in R'$
and $(N, M) \in \text{mult } R$
shows $(\text{image-mset } f \ N, \text{image-mset } f \ M) \in \text{mult } R'$
proof (*insert assms(4), elim multE[OF assms(1)]*)
fix $I \ J \ K$
assume $N = I + K \ M = I + J \ J \neq \{\#\} \forall k \in \text{set-mset } K. \exists j \in \text{set-mset } J.$

$(k, j) \in R$
thus $(\text{image-mset } f \ N, \text{image-mset } f \ M) \in \text{mult } R'$ **using** $\text{assms}(2,3)$
by (intro multI) $(\text{auto}, \text{blast})$
qed

1.4 Incrementality of mult1 and mult

lemma mono-mult1 : mono mult1
unfolding $\text{mono-def mult1-def}$ **by** blast

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

1.5 Well-orders and well-quasi-orders

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

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

lemma *list-splits-empty[simp]*:

list-splits xs = [] \longleftrightarrow xs = []

by (*cases* xs) *simp-all*

lemma *elem-list-splits-append*:

assumes (ys, y, zs) \in set (*list-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-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-splits* xs)

shows y \in set ys

using *elem-list-splits-append[OF assms]* **by** *force*

lemma *list-splits-append*:

list-splits (xs @ ys) = map ($\lambda(xs', x', ys'). (xs', x', ys' @ ys)$) (*list-splits* xs) @
map ($\lambda(xs', x', ys'). (xs @ xs', x', ys')$) (*list-splits* ys)

by (*induct* xs) *auto*

lemma *list-splits-rev*:

list-splits (rev xs) = map ($\lambda(xs, x, ys). (rev ys, x, rev xs)$) (*list-splits* xs)

by (*induct* xs) (*auto simp add: list-splits-append comp-def prod.case-distrib rev-map*)

lemma *list-splits-map*:

list-splits (map f xs) = map ($\lambda(xs, x, ys). (map f xs, f x, map f ys)$) (*list-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*]: $letter-less\ R = \{(a,b). (snd\ a, snd\ b) \in R\}$

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* ($\lambda(xs, y, zs) \Rightarrow adj-msog\ xs\ zs\ (y, [])$) (*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 $x \in \text{set } ys$
using *assms* **by** (*auto dest: elem-list-splits-elim simp: adj-msog-def ms-of-greek-def*)

lemma *ms-of-greek-shorter*:
assumes $(x, t) \in \# \text{ ms-of-greek } s$
shows $\text{length } s > \text{length } t$
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*: $\text{ms-of-greek } (xs @ ys) = \text{image-mset } (\text{adj-msog } [] ys) (\text{ms-of-greek } xs) +$
 $\text{image-mset } (\text{adj-msog } xs []) (\text{ms-of-greek } ys)$
by (*auto simp: ms-of-greek-def list-splits-append multiset.map-comp comp-def prod.case-distrib*)

definition *nest* :: $('a \times 'a) \text{ set} \Rightarrow ('a \text{ greek} \times 'a \text{ greek}) \text{ set} \Rightarrow ('a \text{ greek} \times 'a \text{ greek}) \text{ set}$ **where**
 $[\text{simp}]$: $\text{nest } r \ s = \{(a, b). (\text{ms-of-greek } a, \text{ms-of-greek } b) \in \text{mult } (\text{letter-less } r < *lex* > s)\}$

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

lemma *nest-mono[mono-set]*: $x \subseteq y \Longrightarrow (a, b) \in \text{nest } r \ x \longrightarrow (a, b) \in \text{nest } r \ y$
using *mono-nest[unfolded mono-def, rule-format, of x y r]* **by** *blast*

definition *greek-less* :: $('a \times 'a) \text{ set} \Rightarrow ('a \text{ greek} \times 'a \text{ greek}) \text{ set}$ **where**
 $\text{greek-less } r = \text{lfp } (\text{nest } r)$

lemma *greek-less-unfold*:
 $\text{greek-less } r = \text{nest } r \ (\text{greek-less } r)$
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* $(\text{letter-less } r)$
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* $(\text{nest } r \ R)$

proof –
have *strict-order* (*mult* (*letter-less* *r* $\langle *lex* \rangle$ *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* $\langle *lex* \rangle$ *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 $r \subseteq s$
then **have** *letter-less* *r* $\langle *lex* \rangle$ *R* \subseteq *letter-less* *s* $\langle *lex* \rangle$ *R*
using *mono-letter-less* *mono-lex1* **unfolding** *mono-def* **by** *metis*
then **show** *nest* *r* *R* \subseteq *nest* *s* *R* **using** *mono-mult* **unfolding** *nest-def* *mono-def*
by *blast*
qed

2.4 Involution

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

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

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

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

lemma *inv-letter-mono*[*simp*]:
assumes $(x, y) \in letter\text{-}less\ r$
shows $(inv\text{-}letter\ x, inv\text{-}letter\ y) \in letter\text{-}less\ r$
using *assms* **by** *simp*

definition *inv-greek* :: $'a\ greek \Rightarrow 'a\ greek$ **where**
 $inv\text{-}greek\ s = rev (map\ inv\text{-}letter\ s)$

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

lemma *inv-greek-append*:
 $inv\text{-}greek (s @ t) = inv\text{-}greek\ t @ inv\text{-}greek\ s$
by (*simp add: inv-greek-def*)

definition *inv-msog* :: $('a\ letter \times 'a\ greek)\ multiset \Rightarrow ('a\ letter \times 'a\ greek)\ multiset$ **where**
 $inv\text{-}msog\ M = image\text{-}mset (\lambda(x, t). (inv\text{-}letter\ x, inv\text{-}greek\ t))\ M$

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

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

lemma *inv-greek-mono*:
assumes *trans* r **and** $(s, t) \in greek\text{-}less\ r$
shows $(inv\text{-}greek\ s, inv\text{-}greek\ t) \in greek\text{-}less\ r$
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 $(inv\text{-}msog (ms\text{-}of\text{-}greek\ s), inv\text{-}msog (ms\text{-}of\text{-}greek\ t)) \in mult (letter\text{-}less\ r$
 $\langle *lex* \rangle greek\text{-}less\ r)$
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-empty[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-empty[simp]:
  ([], b) ∈ greek-less r ⟷ b ≠ []
proof
  assume ([], b) ∈ greek-less r
  then show b ≠ [] using greek-less-empty 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** $(s, t) \in \text{greek-less } r$
shows $(p @ s, p @ t) \in \text{greek-less } r$
using *assms* **by** (induct p) (auto simp add: greek-less-cons-mono)

lemma *greek-less-app-mono1*:
assumes *trans r* **and** $(s, t) \in \text{greek-less } r$
shows $(s @ p, t @ p) \in \text{greek-less } r$
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* $(\lambda a b. (a, b): \text{reflcl } (\text{letter-less } r)) a b \implies (a, b) \in \text{reflcl } (\text{greek-less } r)$
proof (induct rule: *list-emb.induct*)
case (*list-emb-Cons a b y*) **thus** ?case
using *trans-greek-less*[*unfolded trans-def*] $\langle \text{trans } r \rangle$
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*] $\langle \text{trans } r \rangle$ *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* $(\lambda a b. (a, b) \in r^=)$ *UNIV*
shows *wqo-on* $(\lambda a b. (a, b) \in (\text{letter-less } r)^=)$ *UNIV*
proof (rule *wqo-on-hom*[of - *id* - *prod-le (=)*] $(\lambda a b. (a, b) \in r^=)$, *unfolded image-id id-apply*)

show $wqo\text{-}on$ ($prod\text{-}le$ ($(=) :: accent \Rightarrow accent \Rightarrow bool$) ($\lambda a b. (a, b) \in r^=$)) $UNIV$
by ($rule$ $dickson[OF$ $finite\text{-}eq\text{-}wqo\text{-}on[OF$ $finite\text{-}accent]$ w , $unfolded$ $UNIV\text{-}Times\text{-}UNIV$])
qed ($insert$ t , $auto$ $simp$: $transp\text{-}on\text{-}def$ $trans\text{-}def$ $prod\text{-}le\text{-}def$)

lemma $wf\text{-}greek\text{-}less$:

assumes wf r **and** $trans$ r
shows wf ($greek\text{-}less$ r)

proof –

obtain q **where** $r \subseteq q$ **and** $well\text{-}order$ q **by** ($metis$ $total\text{-}well\text{-}order\text{-}extension$ $\langle wf$
 $r \rangle$)

define q' **where** $q' = q - Id$

from $\langle well\text{-}order$ $q \rangle$ **have** $reflcl$ $q' = q$

by ($auto$ $simp$ add : $well\text{-}order\text{-}on\text{-}def$ $linear\text{-}order\text{-}on\text{-}def$ $partial\text{-}order\text{-}on\text{-}def$ $pre\text{-}order\text{-}on\text{-}def$

$refl\text{-}on\text{-}def$ $q'\text{-}def$)

from $\langle well\text{-}order$ $q \rangle$ **have** $trans$ q' **and** $irrefl$ q'

unfolding $well\text{-}order\text{-}on\text{-}def$ $linear\text{-}order\text{-}on\text{-}def$ $partial\text{-}order\text{-}on\text{-}def$ $preorder\text{-}on\text{-}def$
 $antisym\text{-}def$

$trans\text{-}def$ $irrefl\text{-}def$ $q'\text{-}def$ **by** $blast+$

from $\langle r \subseteq q \rangle$ $\langle wf$ $r \rangle$ **have** $r \subseteq q'$ **by** ($auto$ $simp$ add : $q'\text{-}def$)

have $wqo\text{-}on$ ($\lambda a b. (a, b) \in (greek\text{-}less$ $q')^=$) $UNIV$

proof ($intro$ $wqo\text{-}on\text{-}hom$ [of ($\lambda a b. (a, b) \in (greek\text{-}less$ $q')^=$) id $UNIV$

$list\text{-}emb$ ($\lambda a b. (a, b) \in (letter\text{-}less$ $q')^=$), $unfolded$ $surj\text{-}id$])

show $transp\text{-}on$ ($\lambda a b. (a, b) \in (greek\text{-}less$ $q')^=$) $UNIV$

using $trans\text{-}greek\text{-}less$ [of q'] **unfolding** $trans\text{-}def$ $transp\text{-}on\text{-}def$ **by** $blast$

next

show $\forall x \in UNIV. \forall y \in UNIV. list\text{-}emb$ ($\lambda a b. (a, b) \in (letter\text{-}less$ $q')^=$) x $y \longrightarrow$
 $(id$ x, id $y) \in (greek\text{-}less$ $q')^=$

using $greek\text{-}embed$ [OF $\langle trans$ $q' \rangle$] **by** $auto$

next

show $wqo\text{-}on$ ($list\text{-}emb$ ($\lambda a b. (a, b) \in (letter\text{-}less$ $q')^=$)) $UNIV$

using $higman$ [OF $wqo\text{-}letter\text{-}less$ [OF $\langle trans$ $q' \rangle$]] $\langle well\text{-}order$ $q \rangle$ $\langle reflcl$ $q' = q \rangle$

by ($auto$ $simp$: $well\text{-}order\text{-}implies\text{-}wqo$)

qed

with $wqo\text{-}on\text{-}imp\text{-}wfp\text{-}on$ [OF $this$] $strict\text{-}order\text{-}strict$ [OF $strict\text{-}order\text{-}greek\text{-}less$]
 $\langle irrefl$ $q' \rangle$ $\langle trans$ $q' \rangle$

have $wfp\text{-}on$ ($\lambda a b. (a, b) \in greek\text{-}less$ q') $UNIV$ **by** $force$

then show $?thesis$

using $mono\text{-}greek\text{-}less$ $\langle r \subseteq q' \rangle$ $wf\text{-}subset$ **unfolding** $wf\text{-}iff\text{-}wfp\text{-}on$ [$symmetric$]
 $mono\text{-}def$ **by** $metis$

qed

2.7 Basic Comparisons

lemma $pairwise\text{-}imp\text{-}mult$:

assumes $N \neq \{\#\}$ **and** $\forall x \in set\text{-}mset$ $M. \exists y \in set\text{-}mset$ $N. (x, y) \in r$

shows $(M, N) \in mult$ r

using $assms$ $one\text{-}step\text{-}implies\text{-}mult$ [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 *set-mset (ms-of-greek as)*
 with *as* *ms-of-greek-elem[of - - as]*
 have (*e*, ((*a,b*),[])) \in *letter-less r <*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* \subseteq *under r a* **and** *b'*: *b' \in {[(Grave,b),[]]}*
and *cs*: *snd* ‘ *set cs* \subseteq *under r a* \cup *under r b* **and** *a'*: *a' \in {[(Acute,a),[]]}*
and *bs*: *snd* ‘ *set bs* \subseteq *under r b*
shows (*as @ b' @ cs @ a' @ bs*, [(*Acute,a*),(*Grave,b*)]) \in *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]*) \in *mult*
 (*letter-less r <*lex*> greek-less r*)
 proof (*intro pairwise-imp-mult*)

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

```

}
moreover {
  fix  $e$  assume  $e \in \text{set-mset } (ms\text{-of-greek } a')$ 
  with  $a'$  singleton-greek-less[ $OF$   $bs$ ] ms-of-greek-elem[ $of$  - -  $a'$ ]
  have (adj-msog ( $as @ b' @ cs$ )  $bs$   $e$ , ( $?A, [?B]$ ))  $\in$  letter-less  $r <*lex*>$ 
greek-less  $r$ 
  by (cases  $e$ ) (fastforce simp: adj-msog-def ms-of-greek-def)
}
moreover {
  fix  $e$  assume  $e \in \text{set-mset } (ms\text{-of-greek } bs)$ 
  with  $bs$  ms-of-greek-elem[ $of$  - -  $bs$ ]
  have (adj-msog ( $as @ b' @ cs @ a'$ ) []  $e$ , ( $?B, [?A]$ ))  $\in$  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  $\forall x \in \text{set-mset } (ms\text{-of-greek } (as @ b' @ cs @ a' @ bs)).$ 
 $\exists y \in \text{set-mset } (ms\text{-of-greek } [?A, ?B]). (x, y) \in \text{letter-less } r <*lex*>$  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  $as \subseteq \text{under } r a \cap \text{under } r b$  and  $b'$ :  $b' \in \{[(Grave, b)], []\}$ 
and  $cs$ : snd ' set  $cs \subseteq \text{under } r b$  and  $a'$ :  $a' = [(Macron, a)]$ 
and  $bs$ : snd ' set  $bs \subseteq \text{under } r b$ 
shows ( $as @ b' @ cs @ a' @ bs$ ,  $[(Macron, a), (Grave, b)]$ )  $\in$  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 [ $(Macron, a)$ ,  $(Grave, b)$ ] - {# ( $(Macron, a)$ ,  $[]$ ) #}
 $\neq$  {#}
  by (auto)

```

```

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

```


r
by (*cases e*) (*fastforce simp: adj-msog-def ms-of-greek-def*)
}
moreover {
fix e **assume** $e \in \text{set-mset } (ms\text{-of-greek } cs)$
with cs *ms-of-greek-elem*[*of - - cs*]
have (*adj-msog* ($as @ b'$) ($a' @ bs$) e , ($?B, [?A]$)) \in *letter-less r* $\langle *lex* \rangle$
greek-less r
by (*cases e*) (*fastforce simp: adj-msog-def under-def*)
}
moreover {
fix e **assume** $e \in \text{set-mset } (ms\text{-of-greek } bs)$
with bs *ms-of-greek-elem*[*of - - bs*]
have (*adj-msog* ($as @ b' @ cs @ a'$) [] e , ($?B, [?A]$)) \in *letter-less r* $\langle *lex* \rangle$
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** $\forall x \in \text{set-mset } (ms\text{-of-greek } (as @ b' @ cs @ a' @ bs) - \{\#(?A, [])\#})$.
 $\exists y \in \text{set-mset } (ms\text{-of-greek } [?A, ?B] - \{\#(?A, [])\#})$. (x, y) \in *letter-less r* $\langle *lex* \rangle$
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* [$?A, ?B$]) \in *mult*
(*letter-less r* $\langle *lex* \rangle$ *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 *rclyff-greek-less2*:

assumes *trans r*
and as : *snd* ' *set* $as \subseteq \text{under } r \ a$ **and** b' : $b' \in \{[(Grave, b)], []\}$
and cs : *snd* ' *set* $cs \subseteq \text{under } r \ a \cup \text{under } r \ b$
shows ($as @ b' @ cs$, [*Macron*, a], (*Grave*, b)) \in *greek-less r*
proof –
let $?A = (Macron, a)$ **and** $?B = (Grave, b)$
have (*ms-of-greek* ($as @ b' @ cs$), *ms-of-greek* [$?A, ?B$]) \in *mult* (*letter-less r*
 $\langle *lex* \rangle$ *greek-less r*)
proof (*intro pairwise-imp-mult*)

{
fix e **assume** $e \in \text{set-mset } (ms\text{-of-greek } as)$
with as *ms-of-greek-elem*[*of - - as*]
have (*adj-msog* [] ($b' @ cs$) e , ($?A, []$)) \in *letter-less r* $\langle *lex* \rangle$ *greek-less r*
by (*cases e*) (*fastforce simp: adj-msog-def under-def*)
}
moreover {

```

fix  $e$  assume  $e \in \text{set-mset } (ms\text{-of-greek } b')$ 
with  $b'$  singleton-greek-less[OF as] ms-of-greek-elem[of - - b']
have (adj-msog as ( $@$   $b'$ )  $\sqcap$   $e$ , ( $?B, [?A]$ ))  $\in$  letter-less r  $\langle *lex* \rangle$  greek-less r
by (cases e) (fastforce simp: adj-msog-def ms-of-greek-def)
}
moreover {
fix  $e$  assume  $e \in \text{set-mset } (ms\text{-of-greek } cs)$ 
with  $cs$  ms-of-greek-elem[of - - cs]
have (adj-msog (as @ b')  $\sqcap$   $e$ , ( $?A, []$ ))  $\in$  letter-less r  $\langle *lex* \rangle$  greek-less r  $\vee$ 
(adj-msog (as @ b')  $\sqcap$   $e$ , ( $?B, [?A]$ ))  $\in$  letter-less r  $\langle *lex* \rangle$  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  $\forall x \in \text{set-mset } (ms\text{-of-greek } (as @ b' @ cs))$ .
 $\exists y \in \text{set-mset } (ms\text{-of-greek } [?A, ?B])$ .  $(x, y) \in$  letter-less r  $\langle *lex* \rangle$  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  $\subseteq$  under r a and  $b'$ :  $b' = [(Macron, b)]$ 
and  $cs$ : snd ' set cs  $\subseteq$  under r a and  $a'$ :  $a' \in \{[(Acute, a)], []\}$ 
and  $bs$ : snd ' set bs  $\subseteq$  under r a  $\cap$  under r b
shows ( $as @ b' @ cs @ a' @ bs$ ,  $[(Acute, a), (Macron, b)]$ )  $\in$  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)]]))  $\in$  greek-less r
apply (rule inv-greek-mono[OF  $\langle trans r \rangle$ ])
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  $\subseteq$  under r a  $\cup$  under r b and  $a'$ :  $a' \in \{[(Acute, a)], []\}$ 
and  $bs$ : snd ' set bs  $\subseteq$  under r b
shows ( $cs @ a' @ bs$ ,  $[(Acute, a), (Macron, b)]$ )  $\in$  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  $\subseteq$  {Grave} set bs  $\subseteq$  {Macron} set cs  $\subseteq$  {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 $conversion' M \equiv ((\bigcup i \in M. R i) \cup (\bigcup i \in M. E i) \cup (\bigcup i \in M. L i)^{-1})^*$
abbreviation $valley' M \equiv (\bigcup i \in M. R i)^* O (\bigcup i \in M. E i)^* O ((\bigcup i \in M. L i)^{-1})^*$

lemma *conversion-to-lconv*:

assumes $(u, v) \in conversion' M$
obtains xs **where** $snd \text{ ' set } xs \subseteq M$ **and** $(u, v) \in lconv xs$
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** $snd p \in M$ **and** $(u', x) \in lstep p$
by (*force split: accent.splits*)
moreover **obtain** xs **where** $snd \text{ ' set } xs \subseteq M$ $(x, v) \in lconv xs$ **by** (*rule step(3)*)
ultimately **show** *?case* **using** *step(4)*[*of p # xs*] **by** *auto*
qed

definition $lpeak :: 'b rel \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b greek \Rightarrow bool$ **where**

$lpeak r a b xs \longleftrightarrow (\exists as b' cs a' bs. snd \text{ ' set } as \subseteq under r a \wedge b' \in \{[(Grave, b)], []\}$
 \wedge
 $snd \text{ ' set } cs \subseteq under r a \cup under r b \wedge a' \in \{[(Acute, a)], []\} \wedge$
 $snd \text{ ' set } bs \subseteq under r b \wedge xs = as @ b' @ cs @ a' @ bs)$

definition $lcliff :: 'b rel \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b greek \Rightarrow bool$ **where**

$lcliff r a b xs \longleftrightarrow (\exists as b' cs a' bs. snd \text{ ' set } as \subseteq under r a \wedge b' = [(Macron, b)]$
 \wedge
 $snd \text{ ' set } cs \subseteq under r a \wedge a' \in \{[(Acute, a)], []\} \wedge$
 $snd \text{ ' set } bs \subseteq under r a \cap under r b \wedge xs = as @ b' @ cs @ a' @ bs) \vee$
 $(\exists cs a' bs. snd \text{ ' set } cs \subseteq under r a \cup under r b \wedge a' \in \{[(Acute, a)], []\} \wedge$
 $snd \text{ ' set } bs \subseteq under r b \wedge xs = cs @ a' @ bs)$

definition $rcliff :: 'b rel \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b greek \Rightarrow bool$ **where**

$rcliff r a b xs \longleftrightarrow (\exists as b' cs a' bs. snd \text{ ' set } as \subseteq under r a \cap under r b \wedge b' \in$
 $\{[(Grave, b)], []\} \wedge$
 $snd \text{ ' set } cs \subseteq under r b \wedge a' = [(Macron, a)] \wedge$
 $snd \text{ ' set } bs \subseteq under r b \wedge xs = as @ b' @ cs @ a' @ bs) \vee$
 $(\exists as b' cs. snd \text{ ' set } as \subseteq under r a \wedge b' \in \{[(Grave, b)], []\} \wedge$
 $snd \text{ ' set } cs \subseteq under r a \cup under r b \wedge xs = as @ b' @ cs)$

lemma *dd-commute-modulo-conv*[*case-names wf trans peak lcliff rcliff*]:

assumes *wf r* **and** *trans r*
and $pk: \bigwedge a b s t u. (s, t) \in L a \Longrightarrow (s, u) \in R b \Longrightarrow \exists xs. lpeak r a b xs \wedge (t, u) \in lconv xs$
and $lc: \bigwedge a b s t u. (s, t) \in L a \Longrightarrow (s, u) \in E b \Longrightarrow \exists xs. lcliff r a b xs \wedge (t, u) \in lconv xs$
and $rc: \bigwedge a b s t u. (s, t) \in (E a)^{-1} \Longrightarrow (s, u) \in R b \Longrightarrow \exists xs. rcliff r a b xs \wedge$

```

(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':  $\bigwedge x y. (x,y) \in lconv as' \implies (x,y) \in$ 
        (∪ a. R a)*
        proof (induct as')
          case (Cons x' xs)
          have  $\bigwedge x y z i. (x,y) \in R i \implies (y,z) \in (\bigcup a. R a)^* \implies (x,z) \in (\bigcup a. R$ 
          a)*
          by (rule rtrancl-trans) auto
          with Cons show ?case by auto
        qed simp
        from bs(1)[unfolded bs(2)] have bs':  $\bigwedge x y. (x,y) \in lconv bs' \implies (x,y) \in$ 
        (∪ a. E a)*
        proof (induct bs')
          case (Cons x' xs)
          have  $\bigwedge x y z i. (x,y) \in E i \implies (y,z) \in (\bigcup a. E a)^* \implies (x,z) \in (\bigcup a. E a)^*$ 
          by (rule rtrancl-trans) auto
          with Cons show ?case by auto
        qed simp
        from cs(1)[unfolded cs(2)] have cs':  $\bigwedge x y. (x,y) \in lconv cs' \implies (x,y) \in$ 
        ((∪ a. L a)-1)*
        proof (induct cs')
          case (Cons x' xs)
          have  $\bigwedge x y z i. (x,y) \in (L i)^{-1} \implies (y,z) \in ((\bigcup a. L a)^{-1})^* \implies (x,z) \in$ 
          ((∪ a. L a)-1)*
          by (rule rtrancl-trans) auto
          with Cons show ?case by auto
        qed simp
      end
    end
  end
end

```

```

    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 $a: (s', t') \in (E a)^{-1}$ **and** $b: (s', u') \in R b$ **and** $q: (u', v) \in lconv q$
obtain js **where** $lp: rcliff r a b$ js **and** $js: (t', u') \in lconv js$ **using** $rc[OF a$
 $b]$ **by** *auto*
from lp **have** $(js, [(Macron, a), (Grave, b)]) \in greek-less r$
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** $(p @ js @ q, xs) \in greek-less r$ **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 $pk: \bigwedge a b s t u. (s, t) \in L a \implies (s, u) \in R b \implies$
 $(t, u) \in conversion' (under r a) O (R b)^= O conversion' (under r a \cup under r$
 $b) O$
 $((L a)^{-1})^= O conversion' (under r b)$
and $lc: \bigwedge a b s t u. (s, t) \in L a \implies (s, u) \in E b \implies$
 $(t, u) \in conversion' (under r a) O E b O conversion' (under r a) O$
 $((L a)^{-1})^= O conversion' (under r a \cap under r b) \vee$
 $(t, u) \in conversion' (under r a \cup under r b) O ((L a)^{-1})^= O conversion'$
 $(under r b)$
and $rc: \bigwedge a b s t u. (s, t) \in (E a)^{-1} \implies (s, u) \in R b \implies$
 $(t, u) \in conversion' (under r a \cap under r b) O (R b)^= O conversion' (under r$
 $b) O$
 $E a O conversion' (under r b) \vee$
 $(t, u) \in conversion' (under r a) O (R b)^= O conversion' (under r a \cup under r$
 $b)$
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  $(t, w) \in \text{conversion}' (\text{under } r \ a)$ 
from conversion-to-lconv[OF this]
obtain as where  $\text{snd } ' \text{ set } as \subseteq \text{under } r \ a \ (t, w) \in \text{lconv}$  as by auto
moreover assume  $(w, x) \in (R \ b)^=$ 
then obtain  $b'$  where  $b' \in \{[(\text{Grave}, b)], []\}$   $(w, x) \in \text{lconv } b'$  by fastforce
moreover assume  $(x, y) \in \text{conversion}' (\text{under } r \ a \cup \text{under } r \ b)$ 
from conversion-to-lconv[OF this]
obtain cs where  $\text{snd } ' \text{ set } cs \subseteq \text{under } r \ a \cup \text{under } r \ b \ (x, y) \in \text{lconv } cs$  by
auto
moreover assume  $(y, z) \in ((L \ a)^{-1})^=$ 
then obtain  $a'$  where  $a' \in \{[(\text{Acute}, a)], []\}$   $(y, z) \in \text{lconv } a'$  by fastforce
moreover assume  $(z, u) \in \text{conversion}' (\text{under } r \ b)$ 
from conversion-to-lconv[OF this]
obtain bs where  $\text{snd } ' \text{ set } bs \subseteq \text{under } r \ b \ (z, u) \in \text{lconv } bs$  by auto
ultimately have  $\exists xs. \text{lpeak } r \ a \ b \ xs \wedge (t, u) \in \text{lconv } xs$ 
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  $(t, w) \in \text{conversion}' (\text{under } r \ a)$ 
  from conversion-to-lconv[OF this]
  obtain as where  $\text{snd } ' \text{ set } as \subseteq \text{under } r \ a \ (t, w) \in \text{lconv}$  as by auto
  moreover assume  $(w, x) \in E \ b$ 
  then obtain  $b'$  where  $b' = [(\text{Macron}, b)]$   $(w, x) \in \text{lconv } b'$  by fastforce
  moreover assume  $(x, y) \in \text{conversion}' (\text{under } r \ a)$ 
  from conversion-to-lconv[OF this]
  obtain cs where  $\text{snd } ' \text{ set } cs \subseteq \text{under } r \ a \ (x, y) \in \text{lconv } cs$  by auto
  moreover assume  $(y, z) \in ((L \ a)^{-1})^=$ 
  then obtain  $a'$  where  $a' \in \{[(\text{Acute}, a)], []\}$   $(y, z) \in \text{lconv } a'$  by fastforce
  moreover assume  $(z, u) \in \text{conversion}' (\text{under } r \ a \cap \text{under } r \ b)$ 
  from conversion-to-lconv[OF this]
  obtain bs where  $\text{snd } ' \text{ set } bs \subseteq \text{under } r \ a \cap \text{under } r \ b \ (z, u) \in \text{lconv } bs$  by
auto
  ultimately have  $\exists xs. \text{lcliff } r \ a \ b \ xs \wedge (t, u) \in \text{lconv } xs$ 
  by (intro exI[of - as @ b' @ cs @ a' @ bs], unfold lconv-append lcliff-def) blast
}
moreover {
  fix  $w \ x$ 
  assume  $(t, w) \in \text{conversion}' (\text{under } r \ a \cup \text{under } r \ b)$ 
  from conversion-to-lconv[OF this]
  obtain cs where  $\text{snd } ' \text{ set } cs \subseteq \text{under } r \ a \cup \text{under } r \ b \ (t, w) \in \text{lconv } cs$  by
auto
  moreover assume  $(w, x) \in ((L \ a)^{-1})^=$ 
  then obtain  $a'$  where  $a' \in \{[(\text{Acute}, a)], []\}$   $(w, x) \in \text{lconv } a'$  by fastforce
  moreover assume  $(x, u) \in \text{conversion}' (\text{under } r \ b)$ 
  from conversion-to-lconv[OF this]

```

```

    obtain bs where snd ' set bs  $\subseteq$  under r b (x, u)  $\in$  lconv bs by auto
    ultimately have  $\exists xs. \text{rliff } r a b xs \wedge (t, u) \in \text{lconv } xs$ 
    by (intro exI[of - cs @ a' @ bs], unfold lconv-append rcliff-def) blast
  }
  ultimately show ?case using lc[OF rcliff] by blast
next
case (rcliff a b s t u)
{
  fix w x y z
  assume (t, w)  $\in$  conversion' (under r a  $\cap$  under r b)
  from conversion-to-lconv[OF this]
  obtain as where snd ' set as  $\subseteq$  under r a  $\cap$  under r b (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 b)
  from conversion-to-lconv[OF this]
  obtain cs where snd ' set cs  $\subseteq$  under r b (x, y)  $\in$  lconv cs by auto
  moreover assume (y, z)  $\in$  E a
  then obtain a' where a' = [(Macron,a)] (y, z)  $\in$  lconv a' by fastforce
  moreover assume (z, u)  $\in$  conversion' (under r b)
  from conversion-to-lconv[OF this]
  obtain bs where snd ' set bs  $\subseteq$  under r b (z, u)  $\in$  lconv bs by auto
  ultimately have  $\exists xs. \text{rliff } r a b xs \wedge (t, u) \in \text{lconv } xs$ 
  by (intro exI[of - as @ b' @ cs @ a' @ bs], unfold lconv-append rcliff-def) blast
}
moreover {
  fix w x
  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, u)  $\in$  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, u)  $\in$  lconv cs by
  auto
  ultimately have  $\exists xs. \text{rliff } r a b xs \wedge (t, u) \in \text{lconv } xs$ 
  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: $\bigwedge i. \text{sym } (E i)$
 and pk: $\bigwedge a b s t u. (s, t) \in L a \implies (s, u) \in L b \implies$

$(t, u) \in \text{conversion}' L L E \text{ (under } r a) O (L b)^= O \text{conversion}' L L E \text{ (under } r a \cup \text{under } r b) O$
 $((L a)^{-1})^= O \text{conversion}' L L E \text{ (under } r b)$
and $cl: \bigwedge a b s t u. (s, t) \in L a \implies (s, u) \in E b \implies$
 $(t, u) \in \text{conversion}' L L E \text{ (under } r a) O E b O \text{conversion}' L L E \text{ (under } r a)$
 O
 $((L a)^{-1})^= O \text{conversion}' L L E \text{ (under } r a \cap \text{under } r b) \vee$
 $(t, u) \in \text{conversion}' L L E \text{ (under } r a \cup \text{under } r b) O ((L a)^{-1})^= O \text{conversion}'$
 $L L E \text{ (under } r b)$
shows $\text{conversion}' L L E UNIV \subseteq \text{valley}' L L E UNIV$
proof (*induct rule: dd-commute-modulo[of r]*)
note $E' = E[\text{unfolded sym-conv-converse-eq}]$
case (*rcliff a b s t u*) **show** ?*case*
using $cl[OF \text{rcliff}(2) \text{rcliff}(1)[\text{unfolded } E], \text{unfolded converse-iff}[of t u, \text{symmetric}]$
by (*auto simp only: E' converse-inward*) (*auto simp only: ac-simps*)
qed fact+

3.2 Commutation and confluence

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

Decreasing diagrams for commutation.

lemma *dd-commute[case-names wf trans peak]*:
assumes *wf r and trans r*
and $pk: \bigwedge a b s t u. (s, t) \in L a \implies (s, u) \in R b \implies$
 $(t, u) \in \text{conversion}'' L R \text{ (under } r a) O (R b)^= O \text{conversion}'' L R \text{ (under } r a$
 $\cup \text{under } r b) O$
 $((L a)^{-1})^= O \text{conversion}'' L R \text{ (under } r b)$
shows *commute* $(\bigcup i. L i) (\bigcup i. R i)$
proof –
have $((\bigcup i. L i)^{-1})^* O (\bigcup i. R i)^* \subseteq \text{conversion}'' L R UNIV$ **by** *regexp*
also have $\dots \subseteq \text{valley}'' L R UNIV$
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 $r q :: 'b \text{ rel}$

assumes *wf r and trans r and trans q and refl q and compat: r O q ⊆ r*
begin

private abbreviation (*input*) $\text{down} :: ('b \Rightarrow 'a \text{ rel}) \Rightarrow ('b \Rightarrow 'a \text{ rel})$ **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 $\langle refl\ q \rangle$ **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 $pk: \bigwedge a\ b\ s\ t\ u. (s, t) \in L\ a \implies (s, u) \in R\ b \implies$
 $(t, u) \in conversion''\ L\ R\ (under\ r\ a)\ O\ (down\ R\ b)^= O\ conversion''\ L\ R\ (under$
 $r\ a \cup under\ r\ b)\ O$
 $((down\ L\ a)^{-1})^= O\ conversion''\ L\ R\ (under\ r\ b)$
shows *commute* $(\bigcup i. L\ i)\ (\bigcup i. R\ i)$
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** $a': (a', a) \in q\ (s, t) \in L\ a'$ **and** $b': (b', b) \in q\ (s, u)$
 $\in R\ b'$
by $(auto\ simp: under-def)$
have $\bigwedge a' a. (a', a) \in q \implies under\ r\ a' \subseteq under\ r\ a$ **using** *compat* **by** $(auto\ simp:$
 $under-def)$
then have $aux1: \bigwedge a' a\ L. (a', a) \in q \implies (\bigcup i \in under\ r\ a'. L\ i) \subseteq (\bigcup i \in under$
 $r\ a. L\ i)$ **by** *auto*
have $aux2: \bigwedge a' a\ L. (a', a) \in q \implies down\ L\ a' \subseteq down\ L\ a$
using $\langle trans\ q \rangle$ **by** $(auto\ simp: under-def\ trans-def)$
have $aux3: \bigwedge a\ L. (\bigcup i \in under\ r\ a. L\ i) \subseteq (\bigcup i \in under\ r\ a. down\ L\ i)$
using $\langle refl\ q \rangle$ **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* $\langle (s, t) \in L\ a' \rangle \langle (s, u) \in R\ b' \rangle$]], *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 Applications*, volume 5117 of *Lecture Notes in Computer Science*, pages 306–320, 2008.