Decreasing-Diagrams-II

By Bertram Felgenhauer

August 21, 2015

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 \textit{mult1} and \textit{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 \textit{greek-less} $r$ ......................... 12
2.6 Well-founded-ness of \textit{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 ................................. 28

1 Preliminaries

\textit{theory} Decreasing-Diagrams-II-Aux
\textit{imports}
..:/Well-Quasi-Orders/Multiset-Extension
..:/Well-Quasi-Orders/Well-Quasi-Orders
\textit{begin}

1.1 Trivialities

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

\textit{abbreviation} strict-order $R \equiv irrefl R \land trans R$

\textit{lemma} order-asym: trans $R \implies asym R = irrefl R$
\textit{unfolding} asym.simps irrefl-def trans-def \textit{by} meson

\textit{lemma} strict-order-strict: strict-order $q \implies strict (\lambda a \ b. (a, b) \in q^=) = (\lambda a \ b. (a, b) \in q)$
\textit{unfolding} trans-def irrefl-def \textit{by} fast

\textit{lemma} mono-lex1: mono $(\lambda r. \text{lex-prod } r \ s)$
\textit{by} (auto simp add: mono-def)

\textit{lemma} mono-lex2: mono $(\text{lex-prod } r)$
\textit{by} (auto simp add: mono-def)

\textit{lemma} irrefl-lex-prod: irrefl $R \implies irrefl S \implies irrefl (R \textltlex S)$
\textit{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 ⇒ bool where
  set-chain C ≡ ∀ x ∈ C. ∀ y ∈ C. x ≤ y ∨ y ≤ x

lemma lfp-chain-induct:
  assumes mono: mono f
  and step: ⋀ x. P x =⇒ P (f x)
  and chain: ⋀ C. set-chain C =⇒ ∀ x ∈ C. P x =⇒ P (Sup C)
  shows P (lfp f)
unfolding lfp-eq-fixp[OF mono]
proof (rule fixp-induct)
  show monotone (op ≤) (op ≤) f using mono unfolding order-class.mono-def monotone-def .
next
  show P (Sup {}) using chain[of {}] by simp
next
  show ccpo.admissible Sup (op ≤) 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: ⋀ R. R ∈ C =⇒ trans R
  shows trans (Sup C)
proof (intro transI)
  fix x y z
  assume ⟨x, y⟩ ∈ Sup C and ⟨y, z⟩ ∈ Sup C
  from ⟨x, y⟩ ∈ Sup C obtain R where R ∈ C and ⟨x, y⟩ ∈ R by blast
  from ⟨y, z⟩ ∈ Sup C obtain S where S ∈ C and ⟨y, z⟩ ∈ S by blast
  from ⟨x ∈ C and ⟨S ∈ C and ⟨set-chain C, have R ∪ S = R ∨ R ∪ S = S
  by blast
  with ⟨R ∈ C⟩ and ⟨S ∈ C⟩ have R ∪ S ∈ C by fastforce
  with ⟨⟨x, y⟩ ∈ R⟩ and ⟨⟨y, z⟩ ∈ S⟩ and trans[of R ∪ S]
  have ⟨x, z⟩ ∈ R ∪ S unfolding trans-def by blast
  with ⟨R ∪ S ∈ C⟩ show ⟨x, z⟩ ∈ ∪ C by blast
qed

lemma asym-Sup-of-chain:
  assumes set-chain C and asym: ⋀ R. R ∈ C =⇒ asym R
shows asym (Sup C)
proof (intro asymI2 notI)
  fix a b
  assume \((a,b) \in \text{Sup} C\) then obtain \(R \in C\) and \((a,b) \in R\) by blast
  assume \((b,a) \in \text{Sup} C\) then obtain \(S \in C\) and \((b,a) \in S\) by blast
  from \(R \in C\) and \(S \in C\) and \(\text{set-chain} C\) have \(R \cup S = R\lor R \cup S = S\)
  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 \(\forall R. \text{strict-order} R \Rightarrow \text{strict-order} (f R)\)
  shows \(\text{strict-order} (\text{lfp} f)\)
proof (intro lfp-chain-induct \[of f \text{ strict-order}\])
  fix C :: \('b \times 'b\) set set
  assume set-chain C and \(\forall R \in C. \text{strict-order} R\)
  from this show \(\text{strict-order} (\text{Sup} C)\) by (metis asym-Sup-of-chain trans-Sup-of-chain
  order-asym)
qed fact+

lemma trans-lfp:
  assumes mono f and \(\forall R. \text{trans} R \Rightarrow \text{trans} (f R)\)
  shows \(\text{trans} (\text{lfp} f)\)
by (metis lfp-chain-induct \[of f \text{ trans}\] assms trans-Sup-of-chain
  order-asym)
end

1.3 Multiset extension

lemma mulex-iff-mult: mulex \(r M N \iff (M,N) \in \text{mult} \{ (M,N) . r M N \}\)
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 ≠ \{\#\} \(\forall k \in \text{set-of} K. \exists j \in \text{set-of} J. (k,j) \in r\)
  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 ≠ \{\#\} \(\forall k \in \text{set-of} K. \exists j \in \text{set-of} J. (k,j) \in r\)
using mult-implies-one-step[of assms] by blast

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

by (metis mult-def not-less-empty trancl_cases)

lemma mult-singleton[simp]: \((x, y) \in r \implies (\{#x#\}, \{#y#\}) \in \text{mult } r\)

using mult1-singleton[of x y r] by (auto simp add: mult-def)

lemma empty-mult[simp]: \((\emptyset, N) \in \text{mult } R \iff N \neq \emptyset\)

using empty-mulex-on[of N UNIV \(\lambda M N. (M, N) \in R\)] by (auto simp add: mulex-iff-mult)

lemma trans-mult: trans \((\text{mult } R)\)

unfolding mult-def by simp

lemma strict-order-mult:

assumes irrefl R and trans R

shows irrefl (\text{mult } R) and trans (\text{mult } R)

proof –

show irrefl (\text{mult } R) unfolding irrefl-def

proof (intro allI notI, elim multE[OF \(\langle\text{trans } R\rangle\)])

fix \(M, I, J, K\)

assume \(M = I + J\) and \(M = I + K\) \(J \neq \emptyset\) and \(*: \forall k \in \text{set-of } K. \exists j \in \text{set-of } J\)

have \(\text{finite } (\text{set-of } J)\) by simp

then have \(\text{set-of } J = \emptyset\) using \(*\) unfolding \(\langle\text{J } = \emptyset\rangle\)

by (induct rule: finite_induct)

thus \(\langle\text{False } using \langle\text{J } \neq \emptyset\rangle\rangle\) by simp

qed

qed (simp add: trans-mult)

lemma mult-of-image-mset:

assumes trans R and trans R'

and \(\forall x y. x \in \text{set-of } N \implies y \in \text{set-of } M \implies (x, y) \in R \implies (f x, f y) \in R'\)

and \((N, M) \in \text{mult } R\)

shows (image-mset \(f N\), image-mset \(f M\)) \(\in \text{mult } R'\)

proof (intro assms(4), elim multE[OF assms(1)])

fix \(I, J, K\)

assume \(N = I + K\) \(M = I + J\) \(J \neq \emptyset\) \(\forall k \in \text{set-of } K. \exists j \in \text{set-of } J. (k, j) \in R\)

thus (image-mset \(f N\), image-mset \(f M\)) \(\in \text{mult } R'\) using assms(2,3)

by (intro mult1, auto simp del: mem-set-of-iff) blast

qed

1.4 Incrementality of \text{mult1} and \text{mult}

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

lemma mono-mult: mono mult
unfolding mono-mult def mult-def
proof (intro allI impI subsetI)
fix R S :: 'a rel and x
assume R ⊆ S and x ∈ (mult1 R)⁺
then show x ∈ (mult1 S)⁺
using mono-mult[introduced 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 ←→ wfp-on (λ a b. (a, b) ∈ 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-on (λ a b. (a, b) ∈ r) UNIV
proof (intro wqo-onI almost-full-onI)
show transp-on (λ a b. (a, b) ∈ r) UNIV using asms
by (auto simp only: well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def
trans-def transp-on-def)
next
fix f :: nat ⇒ 'a
show good (λ a b. (a, b) ∈ r) f
using asms 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 ∉ UNIV ∨ (f (Suc x), f x) ∉ r - Id
then have (f x, f (Suc x)) ∈ 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 (λ a b. (a, b) ∈ r) f by (auto simp: good-def)
qed
qed

1.6 Splitting lists into prefix, element, and suffix

fun list-splits :: 'a list ⇒ ('a list × 'a × 'a list) list where
list-splits [] = []
| list-splits (x # xs) = ([], x, xs) # map (λ(xs, x', xs'). (x # xs, x', xs')) (list-splits xs)

lemma list-splits_empty[simp]:
list-splits xs = [] ⟷ xs = []
by \((\text{cases } xs)\) simp-all

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

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

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

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

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

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

end

2 Decreasing Diagrams

theory Decreasing-Diagrams-II
imports
  Decreasing-Diagrams-II-Aux
  ~/src/HOL/Cardinals/Wellorder-Extension
  ../Abstract-Rewriting/Abstract-Rewriting
begin

2.1 Greek accents

datatype accent = Acute | Grave | Macron

lemma \textit{UNIV-accent}:
  \(\text{UNIV } = \{ \text{Acute, Grave, Macron} \}\)
  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 ⇒ zs @ ys | Macron ⇒ ys)

definition ms-of-greek :: 'a greek ⇒ ('a letter × 'a greek) multiset where 
ms-of-greek as = multiset-of
(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-of (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-of-map multiset.map-comp comp-def
comp-def prod.case-distrib)

definition nest :: ('a × 'a) 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[of x y r] unfolding 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 impl 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 \((x, y) \in \text{letter-less } r\)
  shows \((\text{inv-letter } x, \text{inv-letter } y) \in \text{letter-less } r\)
using assms by simp

definition inv-greek :: \('
'\) greek \Rightarrow \('
'\) 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 \((s \circ t)\) = inv-greek t \circ inv-greek s
by (simp add: inv-greek-def)

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

lemma inv-msog-invol[simp]:
  inv-msog (inv-msog M) = M
by (simp add: inv-msog-def multiset.map-comp comp-def case-prod-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
multiset-of-map
  multiset.map-comp multiset.of-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 \((s, t) \in \text{greek-less } r\)
  shows \((\text{inv-greek } s, \text{inv-greek } t) \in \text{greek-less } r\)
using assms
proof (induct length s + length t arbitrary: s t rule: less-induct)
  note * = trans-lex-prod[OF trans-letter-less[OF trans-letter-less[OF r]]]
  case (less s t)
  have \((\text{inv-msog } (\text{ms-of-greek } s), \text{inv-msog } (\text{ms-of-greek } t)) \in \text{mult } (\text{letter-less } r <\text{lex}=> \text{greek-less } r)\)
  unfolding inv-msog-def
  proof (induct rule: mult-of-image-mset[OF *])
    case (1 x y) thus ?case
    by (auto intro: less(1)[OF OF'] split: prod.splits dest!: ms-of-greek-shorter)
  next
case 2 thus ?case using less(3) by (subst asm greek-less-unfold) simp
qed
thus 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]:
  \((a,[])\) \in greek-less r \iff False
by (subst greek-less-unfold) (auto simp: ms-of-greek-def)

lemma greek-less-nonempty:
  assumes \(b \neq []\)
  shows \((a,b)\) \in greek-less r \iff \(a,b) \in nest r\) (greek-less r)
by (subst greek-less-unfold) simp

lemma greek-less-lempty[simp]:
  \(([] ,b)\) \in greek-less r \iff \(b \neq []\)
proof
  assume \(([] ,b)\) \in greek-less r
  then show \(b \neq []\) using greek-less-rempty by fast
next
  assume \(b \neq []\)
  then show \(([] ,b)\) \in greek-less r
 unfolding greek-less-nonempty[of ⟨b \neq []⟩] by (simp add: ms-of-greek-def)
qed

lemma greek-less-singleton:
  \((a , b)\) \in letter-less r \implies ([a] , [b]) \in 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 \((x \# s)\) \in\{\# adj-msog [] s \((x,[])\) \#\} + image-mset \((adj-msog [x] []\) \in\ms-of-greek s\)
  \(\)\) \in\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 \((s , t)\) \in greek-less r \implies \((x \# s , x \# t)\) \in 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\) \in\ms-of-greek r\)
    proof (rule mult-on-union, induct rule: mult-of-image-mset[of \(* *\)])
      case \((1 x)\) 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 \\
      \in
      (mult (letter-less r <*lex*> greek-less r))=
      by (auto simp: adj-msog-def less split: accent.splits intro: mult-on-union' mult-singleton)
}
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 greek-less r
shows \( p @ s, p @ t \) \in 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 greek-less r
shows \( s @ p, t @ p \) \in 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)

\[13\]

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

lemma greek-embed:
assumes trans r
shows list-emb \( \lambda a b. (a, b) \): reflcl (letter-less r)) a b \implies (a, b) \in 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-mono[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-mono[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 \( \lambda a b. (a, b) \in \langle letter-less r \rangle^=\) UNIV
proof (rule wqo-on-hom[of - id - prod-le (op =) \( \lambda a b. (a, b) \in r^=\), unfolded image-id id-apply])
  show wqo-on \( \langle prod-le (op = :: accent \Rightarrow accent \Rightarrow bool) \rangle \( \lambda a b. (a, b) \in r^=\)) UNIV
  by (rule dickson[OF finite-eq-wqo-on[OF finite-accent] w, unfolded UNIV-Times-UNIV])
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⟩)
  def 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 antisym-def trans-def irrefl-def q′-def by blast
  from ⟨r ⊆ q⟩ ⟨wf r⟩ have r ⊆ q′ by (auto simp add: q′-def)
  have wqo-on (λ a b. (a, b) ∈ (greek-less q′) = UNIV
  proof (intro wqo-on-hom[of (λ a b. (a, b) ∈ (greek-less q′) =) id UNIV
    list-emb (λ a b. (a, b) ∈ (letter-less q′) =, unfolded surj-id])
  show transp-on (λ a b. (a, b) ∈ (greek-less q′) =) UNIV
  using trans-greek-less[of q′] unfolding trans-def transp-on-def by blast
  next
    show ∀ x∈UNIV. ∀ y∈UNIV. list-emb (λ a b. (a, b) ∈ (letter-less q′) =) x y —→ (id x, id y) ∈ (greek-less q′) =
    using greek-embed[OF ⟨trans q′⟩] by auto
  next
    show wqo-on (list-emb (λ a b. (a, b) ∈ (letter-less q′) =)) UNIV
  using higman[OF wqo-letter-less[OF ⟨trans q′⟩]] ⟨well-order q⟩ ⟨refcl q′ = q⟩
  by (auto simp: well-order-implies-wqa)
  qed
  with wqo-on-imp-wfp-on[OF this] strict-order-strict[OF strict-order-greek-less]
  ⟨irrefl q′ ⟨trans q′⟩⟩ have wfp-on (λ a b. (a, b) ∈ (greek-less q′) =) UNIV by force
  then show ?thesis
  using mono-greek-less ⟨r ⊆ q′⟩ wf-subset unfolding wf-iff-wfp-on[symmetric] mono-def by metis
qed

2.7 Basic Comparisons

lemma pairwise-imp-mult:
  assumes trans r and N ≠ {#} and ∀ x ∈ set-of M. ∃ y ∈ set-of N. (x, y) ∈ r
  shows (M, N) ∈ mult r
using assms one-step-implies-mult[of - - - {#}] by auto

lemma singleton-greek-less:
  assumes trans r and as: snd ' set as ⊆ under r b
  shows (as, [(a, b)]) ∈ greek-less r

qed
proof –
\{
  \text{fix } e \text{ assume } e \in \text{set-of (ms-of-greek as)}
  \text{with as ms-of-greek-elem[of - as]}
  \text{have (e, [(a,b),[]])} \in \text{letter-less r }<\text{lex}> \text{ greek-less r}
\}
\text{moreover have ms-of-greek [(a,b)] = \{# ((a,b),[]) #\}}
\text{by (auto simp: ms-of-greek-def split: accent.splits)}
\text{ultimately show } \text{thesis using trans-letter-less[OF (trans r)]}
\text{by (subst greek-less-unfold) (auto intro: pairwise-imp-mult)}
\text{qed}

\text{lemma peak-greek-less:}
\text{assumes trans r}
\text{and as: snd'} set as \subseteq \text{under r a and b': b' }\in \{[(\text{Acute},b)],[]\}
\text{and cs: snd'} set cs \subseteq \text{under r a }\cup \text{under r b and a': a' }\in \{[(\text{Acute},a)],[]\}
\text{and bs: snd'} set bs \subseteq \text{under r b}
\text{shows (as @ b' @ cs @ a' @ bs, [(\text{Acute},a),(\text{Acute},b)])} \in \text{greek-less r}
proof –
\text{let } ?A = (\text{Acute},a) \text{ and } ?B = (\text{Acute},b)
\text{have (ms-of-greek (as @ b' @ cs @ a' @ bs), ms-of-greek [?A,?B])} \in \text{mult (letter-less r }<\text{lex}> \text{ greek-less r)}
\text{proof (intro pairwise-imp-mult)}
\text{show trans (letter-less r }<\text{lex}> \text{ greek-less r)}
\text{using (trans r) trans-letter-less by auto}
next
\{
  \text{fix } e \text{ assume } e \in \text{set-of (ms-of-greek as)}
  \text{with as ms-of-greek-elem[of - as]}
  \text{have (adj-msog [] (b' @ cs @ a' @ bs) e, (?A,?[B])))} \in \text{letter-less r }<\text{lex}> \text{ greek-less r}
\text{by (cases e) (fastforce simp: adj-msog-def under-def)}
\}
\text{moreover {}
  \text{fix } e \text{ assume } e \in \text{set-of (ms-of-greek b')} 
  \text{with b' singleton-greek-less[OF (trans r) as] ms-of-greek-elem[of - b']}
  \text{have (adj-msog as (cs @ a' @ bs) e, (?B,?[A])))} \in \text{letter-less r }<\text{lex}> \text{ greek-less r}
\text{by (cases e) (fastforce simp: adj-msog-def ms-of-greek-def)}
\}
\text{moreover {}
  \text{fix } e \text{ assume } e \in \text{set-of (ms-of-greek cs) }
  \text{with cs ms-of-greek-elem[of - cs]}
  \text{have (adj-msog (as @ b') (a' @ bs) e, (?A,?[B])))} \in \text{letter-less r }<\text{lex}> \text{ greek-less r} \lor
  \text{(adj-msog (as @ b') (a' @ bs) e, (?B,?[A])))} \in \text{letter-less r }<\text{lex}> \text{ greek-less r}
\}
by (cases e) (fastforce simp: adj-msog-def under-def)

moreover {
  fix e assume e ∈ set-of (ms-of-greek a')
  with a' singleton-greek-less[OF (trans r) bs] ms-of-greek-elem[of - - a']

  have (adj-msog (as @ b' @ cs) bs e, (?A,]?B)) ∈ letter-less r <*lex*>

by (cases e) (fastforce simp: adj-msog-def ms-of-greek-def)

moreover {
  fix e assume e ∈ set-of (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*>

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-of (ms-of-greek (as @ b' @ cs @ a' @ bs)).
  ∃ y ∈ set-of (ms-of-greek [?A,]?B). (x, y) ∈ letter-less r <*lex*>

by (auto simp: msog-append) blast

qed (auto simp: ms-of-greek-def)

then show ?thesis by (subst greek-less-unfold) auto

qed

lemma reliff-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,]?B) #} ms-of-greek [?A] = {# (?A,] #} #
by (simp add: adj-msog-def ms-of-greek-def)

  then have **: ms-of-greek [(Macron, a), (Grave, b)] ∈ {#(Macron, a), [] #} #
≠ {#}
by (auto)

  have ***: trans (letter-less r <*lex*> greek-less r)
using (trans r) trans-letter-less by auto

{ fix e assume e ∈ set-of (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*>

by (cases e) (force simp: adj-msog-def under-def)

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

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

moreover {
  fix $e$ assume $e \in set$-of (ms-of-greek bs)
  with $bs$ ms-of-greek-elem[af - - bs]
  have (adj-msog (as @ $b' @ cs @ a'$) $e$, (?$B,?[?A]?>)) $\in$ letter-less r $<$*lex*> greek-less
  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 $\forall x \in set$-of (ms-of-greek (as @ $b' @ cs @ a'$ @ bs)) $-$ $\{#(??A,[])\}$.
$\exists y \in set$-of (ms-of-greek [?A,?B] $-$ $\{#(??A,[])\}$). ($x$, $y$) $\in$ letter-less r $<$*lex*> greek-less

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 $<$*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 reliff-greek-less2:
assumes trans r
and 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
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 $<$*lex*> greek-less r)
  proof (intro pairwise-imp-mult)
    show trans (letter-less r $<$*lex*> greek-less r)
    using (trans r) trans-letter-less by auto
next
fix e assume e ∈ set-of (ms-of-greek as)
  with as ms-of-greek-elem[of - - as]
  have (adj-msog [] (b' @ cs) e, (?A,[])) ∈ letter-less r <\text{lex}*> greek-less r
  by (cases e) (fastforce simp: adj-msog-def under-def)
}
moreover {
  fix e assume e ∈ set-of (ms-of-greek b')
  with b' singleton-greek-less[OF (trans r) as] ms-of-greek-elem[of - - b']
  have (adj-msog as (cs) e, (?B,[])) ∈ letter-less r <\text{lex}*> greek-less r
  by (cases e) (fastforce simp: adj-msog-def ms-of-greek-def)
}
moreover {
  fix e assume e ∈ set-of (ms-of-greek cs)
  with cs ms-of-greek-elem[of - - cs]
  have (adj-msog (as @ b') [] e, (?A,[])) ∈ letter-less r <\text{lex}*> greek-less r ∨
    (adj-msog (as @ b') [] e, (?B,[])) ∈ letter-less r <\text{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-of (ms-of-greek (as @ b' @ cs)),
  ∃y∈set-of (ms-of-greek (?A,?B)). (x, y) ∈ letter-less r <\text{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-monot 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 \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)
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
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

20
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 xs ⊆ M and (u, v) ∈ lconv xs
  using assms
proof (induct arbitrary: thesis rule: converse-trancl-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 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 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 a ∪ under r b ∧ xs = as @ b' @ cs @ a' @ bs)

definition lcliff :: 'b rel ⇒ '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 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 ∧ a' = [(Macron,b)] ∧
  snd ' set cs ⊆ under r b ∧ xs = as @ b' @ cs @ a' @ bs) ∨
(∃ as b' cs a' bs. snd ' set as ⊆ under r a ∩ 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]:

21
assumes \( \text{wf} \) \( r \) and \( \text{trans} \) \( r \)
and \( pk: \forall a \ b \ s \ t \ u. \ (s, t) \in L \ a \implies (s, u) \in R \ b \implies \exists \xs. \ lpeak \ r \ a \ b \ \xs \land (t, u) \in lconv \ xs \)
and \( lc: \forall a \ b \ s \ t \ u. \ (s, t) \in L \ a \implies (s, u) \in E \ b \implies \exists \xs. \ lcliff \ r \ a \ b \ \xs \land (t, u) \in lconv \ xs \)
and \( rc: \forall a \ b \ s \ t \ u. \ (s, t) \in (E \ a)^{-1} \implies (s, u) \in R \ b \implies \exists \xs. \ rcliff \ r \ a \ b \ \xs \land (t, u) \in lconv \ xs \)
shows conversion' \( \text{UNIV} \subseteq \text{valley'} \text{UNIV} \)
proof (intro subrelI)
  fix \( u \ v \)
assume \((u,v) \in \text{conversion'} \text{UNIV} \)
then obtain \( \xs \) where \((u, v) \in lconv \xs \) by (auto intro: conversion-to-lconv[of \( u \ v \)])
then show \((u, v) \in \text{valley'} \text{UNIV} \)
proof (induct \( \xs \) rule: \text{wf-induct}[\text{of \( \text{greek-less} \) \( r \)]}
  case 1 thus ?case using \text{of \( \text{greek-less} \) \{OF \( \text{wf} \) \( r \) \( \text{of \( \text{trans} \) \( r \) \} \)
next
  case (2 \( \xs \)) show ?case
proof (rule conversion-join-or-peak-or-cliff[of map fst \( \xs \)])
fix \( \text{as} \ \text{bs} \ \text{cs} \)
assume \( \ast: \text{set} \ \text{as} \subseteq \{\text{Grave}\} \ \text{set} \ \text{bs} \subseteq \{\text{Macron}\} \ \text{set} \ \text{cs} \subseteq \{\text{Acute}\} \ \text{map} \ \text{fst} \ \text{xs} \)
\( as = \text{as} @ \text{bs} @ \text{cs} \)
then show \((u, v) \in \text{valley'} \text{UNIV} \)
proof (elim map-eq-append-splits)
fix \( \text{as'} \ \text{bs'} \ \text{cs'} \)
assume \( \text{as}: \text{set} \ \text{as} \subseteq \{\text{Grave}\} \ \text{as} = \text{map} \ \text{fst} \ \text{as'} \) and
\( \text{bs}: \text{set} \ \text{bs} \subseteq \{\text{Macron}\} \ \text{bs} = \text{map} \ \text{fst} \ \text{bs'} \) and
\( \text{cs}: \text{set} \ \text{cs} \subseteq \{\text{Acute}\} \ \text{cs} = \text{map} \ \text{fst} \ \text{cs'} \) and
\( \text{xs}: \text{xs} = \text{as'} \@ \text{bs'} \@ \text{cs'} = \text{bs'} \@ \text{cs'} \)
from \( \text{as}(1)[\text{unfolded as}(2)] \) have \( \text{as'}: \forall x \ y. \ (x,y) \in lconv \text{as'} \implies (x,y) \in (\bigcup a. \ R a)^* \)
proof (induct \( \text{as'} \))
  case (Cons \( x' \) \( \xs \))
  have \( \forall x \ y \ z \ i. \ (x,y) \in R \ i \implies (y,z) \in (\bigcup a. \ R a)^* \implies (x,z) \in (\bigcup a. \ R a)^* \)
  by (rule rtrancl-trans) auto
  with \( \text{Cons} \text{ show ?case by auto} \)
qed simp
from \( \text{bs}(1)[\text{unfolded bs}(2)] \) have \( \text{bs'}: \forall x \ y. \ (x,y) \in lconv \text{bs'} \implies (x,y) \in (\bigcup a. \ E a)^* \)
proof (induct \( \text{bs'} \))
  case (Cons \( x' \) \( \xs \))
  have \( \forall x \ y \ z \ i. \ (x,y) \in E \ i \implies (y,z) \in (\bigcup a. \ E a)^* \implies (x,z) \in (\bigcup a. \ E a)^* \)
  by (rule rtrancl-trans) auto
  with \( \text{Cons} \text{ show ?case by auto} \)
qed simp
from \( \text{cs}(1)[\text{unfolded cs}(2)] \) have \( \text{cs'}: \forall x \ y. \ (x,y) \in lconv \text{cs'} \implies (x,y) \in ((\bigcup a. \ L a)^{-1})^* \)

22
proof (induct cs')
  case (Cons x' xs)
    have \( \forall x y z i. (x,y) \in (L i)^{-1} \rightarrow (y,z) \in ((\bigcup a. L a)^{-1})^* \rightarrow (x,z) \in ((\bigcup a. L a)^{-1})^* \)
      by (rule rtrancl-trans) auto
    with Cons show \( ?case \) by auto
qed simp
next
fix as bs
assume \( * \): \( \text{map fst } xs = \text{as } @ ([\text{Acute}] @ [\text{Grave}]) @ bs \)

\{ fix p a b q t' s' u' 
    assume \( xs: xs = p @ ([\text{Acute},a],[\text{Grave},b]) @ q \text{ and } (u,t') \in \text{lconv } p \)
    and \( a: (s',t') \in L a \text{ and } b: (s',u') \in R b \text{ and } (u',v) \in \text{lconv } q \)
    obtain js where \( lp: lpeak r a b \) \( \) and \( js: (t',u') \in \text{lconv } js \) using \( pk[\text{OF } a b] \) by auto
from \( lp \text{ have } (js, ([\text{Acute},a],[\text{Grave},b])) \in \text{greek-less } r \)
unfolding \( lpeak-def \) using \( \text{peak-greek-less[OF } \text{trans } r, \text{of } a - b \) by fastforce
then have \( (p @ js @ q, xs) \in \text{greek-less } r \) unfolding \( xs \)
by \( (\text {intro greek-less-app-mono1 greek-less-app-mono2 } \text{trans } r) \) auto
moreover have \( (u, v) \in \text{lconv } (p @ js @ q) \)
using \( p q js \) by auto
ultimately have \( (u, v) \in \text{valley' } UNIV \) using \( 2(1) \) by blast
\}
with \( * \) show \( (u, v) \in \text{valley' } UNIV \) using \( 2(2) \)
by \( (\text{auto elim!: map-eq-append-splits relcompEpair simp del: append.simps}) \)
simp
next
fix as bs
assume \( * \): \( \text{map fst } xs = \text{as } @ ([\text{Acute}] @ [\text{Macron}]) @ bs \)

\{ fix p a b q t' s' u' 
    assume \( xs: xs = p @ ([\text{Acute},a],[\text{Macron},b]) @ q \text{ and } (u,t') \in \text{lconv } p \)
    and \( a: (s',t') \in L a \text{ and } b: (s',u') \in E b \text{ and } (u',v) \in \text{lconv } q \)
    obtain js where \( lp: lcliff r a b \) \( \) and \( js: (t',u') \in \text{lconv } js \) using \( le[\text{OF } a b] \) by auto
from \( lp \text{ have } (js, ([\text{Acute},a],[\text{Macron},b])) \in \text{greek-less } r \)
unfolding \( lcliff-def \)
using \( \text{lcliff-greek-less1[OF } \text{trans } r, \text{of } a - b \) \text{lcliff-greek-less2[OF } \text{trans } r, \text{of } a - b \) by fastforce
then have \( (p @ js @ q, xs) \in \text{greek-less } r \) unfolding \( xs \)
by \( (\text{intro greek-less-app-mono1 greek-less-app-mono2 } \text{trans } r) \) auto
moreover have \( (u, v) \in \text{lconv } (p @ js @ q) \)
using \( p q js \) by auto
ultimately have \( (u, v) \in \text{valley' } UNIV \) using \( 2(1) \) by blast
\}
with \( * \) show \( (u, v) \in \text{valley' } UNIV \) using \( 2(2) \)
by \((\text{auto elim!}: \text{map-eq-append-splits relcompEpair simp del: append.simps})\)

\[\text{simp}\]

\[\text{next}\]

\[\text{fix as bs assume *: map fst xs = as @} ([\text{Macron}@]@[\text{Grave}]) @ bs\]

\[\{\]

\[\text{fix p a q t’ s’ u’}\]

\[\text{assume xs: xs = p @} ([\text{Macron},a],(\text{Grave},b)]) @ q \text{ and p: } (u,t’) \in \text{lconv p}\]

\[\text{and a: } (s’,t’) \in (E a)^{-1} \text{ and b: } (s’,u’) \in R b \text{ and q: } (u’,v) \in \text{lconv q}\]

\[\text{obtain js where lp: rcliff r a b js and js: } (t’,u’) \in \text{lconv js using } rc[\text{OF a b}]\]

\[\text{by auto}\]

\[\text{from lp have } (js, ([\text{Macron},a],(\text{Grave},b))) \in \text{greek-less r}\]

\[\text{unfolding rcliff-def}\]

\[\text{using rcliff-greek-less1[OF } \langle \text{trans r }\rangle, \text{ of - a b }\text{ rcliff-greek-less2[OF } \langle \text{trans r }\rangle, \text{ of - a - b}\]

\[\text{by fastforce}\]

\[\text{then have } (p @ js @ q, xs) \in \text{greek-less r unfolding xs}\]

\[\text{by } (\text{intro greek-less-app-mono1 greek-less-app-mono2 } \langle \text{trans r }\rangle) \text{ auto}\]

\[\text{moreover have } (u, v) \in \text{lconv } (p @ js @ q)\]

\[\text{using p q js by auto}\]

\[\text{ultimately have } (u, v) \in \text{valley’ UNIV using 2(1) by blast}\]

\[\}\]

\[\text{with * show } (u, v) \in \text{valley’ UNIV using 2(2)}\]

\[\text{by } (\text{auto elim!}: \text{map-eq-append-splits relcompEpair simp del: append.simps})\]

\[\text{simp}\]

\[\text{qed}\]

\[\text{qed}\]

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** \(w f r \text{ and } \text{trans r}\)

**And pk:** \(\forall a b s t u. \ (s, t) \in L a \Rightarrow (s, u) \in R b \Rightarrow\)

\[(t, u) \in \text{conversion’ (under r a) } O \ (R b)^{-1} \text{ O conversion’ (under r a } \bigcup \text{ under r b) } O\]

\[(L a)^{-1} = O \text{ conversion’ (under r b) } O\]

**And lc:** \(\forall a b s t u. \ (s, t) \in L a \Rightarrow (s, u) \in E b \Rightarrow\)

\[(t, u) \in \text{conversion’ (under r a) } O E b \text{ O conversion’ (under r a) } O\]

\[(L a)^{-1} = O \text{ conversion’ (under r a } \bigcap \text{ under r b) } \bigvee\]

\[(t, u) \in \text{conversion’ (under r a } \bigcup \text{ under r b) } O ((L a )^{-1} = O \text{ conversion’ (under r b) } O\]

**And rc:** \(\forall a b s t u. \ (s, t) \in (E a)^{-1} \Rightarrow (s, u) \in R b \Rightarrow\)

\[(t, u) \in \text{conversion’ (under r a } \bigcap \text{ under r b) } O (R b)^{-1} \text{ O conversion’ (under r b) } O\]

\[E a O \text{ conversion’ (under r b) } \bigvee\]

24
\[(t, u) \in \text{conversion}' (\text{under } r \ a) \ O (R b)\] = \ O \text{conversion}' (\text{under } r \ a \cup \text{under } r \ b)\]

shows \ \text{conversion}' \ \text{UNIV} \subseteq \text{valley}' \ \text{UNIV}

proof (cases rule: \text{dd-commute-modulo-conv[of } r])

case (\text{peak } a \ b \ s \ t \ u)

\{
\begin{align*}
\text{fix } w \ x \ y \ z & \\
\text{assume } (t, w) \in \text{conversion}' (\text{under } r \ a) & \\
\text{from } \text{conversion-to-locale}[\text{OF this}] & \\
\text{obtain as where } \text{snd}' \ \text{set as} \subseteq \text{under } r \ a \ (t, w) \in \text{lconv as} \ \text{by auto} & \\
\text{moreover assume } (w, x) \in (R b)\text{=} & \\
\text{then obtain } b' \ \text{where } b' \in \{[\text{Grave},b]\},\ldots & \\
\text{moreover assume } (x, y) \in \text{conversion}' (\text{under } r \ a \cup \text{under } r \ b) & \\
\text{from } \text{conversion-to-locale}[\text{OF this}] & \\
\text{obtain } \text{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} & \\
\text{moreover assume } (y, z) \in ((L a)^{-1})\text{=} & \\
\text{then obtain } a' \ \text{where } a' \in \{[\text{Acute},a]\},\ldots & \\
\text{moreover assume } (z, u) \in \text{conversion}' (\text{under } r \ b) & \\
\text{from } \text{conversion-to-locale}[\text{OF this}] & \\
\text{obtain } \text{bs where } \text{snd}' \ \text{set bs} \subseteq \text{under } r \ b (z, u) \in \text{lconv bs by auto} & \\
\text{ultimately have } \exists \text{xs. lpeak } r \ a \ b \ \text{xs} \land (t, u) \in \text{lconv xs} & \\
\text{by } (\text{intro exI[of } - \text{@ } b' @ \ \text{cs} @ a' @ \text{bs], unfold lconv-append lpeak-def ] blast} & \\
\end{align*}
\}

then show \ ?case using \ \text{pk}[\text{OF peak}] \ \text{by blast}

next

case (\text{lcliff } a \ b \ s \ t \ u)

\{
\begin{align*}
\text{fix } w \ x \ y \ z & \\
\text{assume } (t, w) \in \text{conversion}' (\text{under } r \ a) & \\
\text{from } \text{conversion-to-locale}[\text{OF this}] & \\
\text{obtain as where } \text{snd}' \ \text{set as} \subseteq \text{under } r \ a \ (t, w) \in \text{lconv as} \ \text{by auto} & \\
\text{moreover assume } (w, x) \in E b & \\
\text{then obtain } b' \ \text{where } b' = [[\text{Macron},b]] (w, x) \in \text{lconv b'} \ \text{by fastforce} & \\
\text{moreover assume } (x, y) \in \text{conversion}' (\text{under } r \ a) & \\
\text{from } \text{conversion-to-locale}[\text{OF this}] & \\
\text{obtain } \text{cs where } \text{snd}' \ \text{set cs} \subseteq \text{under } r \ a (x, y) \in \text{lconv cs by auto} & \\
\text{moreover assume } (y, z) \in ((L a)^{-1})\text{=} & \\
\text{then obtain } a' \ \text{where } a' \in \{[\text{Acute},a]\},\ldots & \\
\text{moreover assume } (z, u) \in \text{conversion}' (\text{under } r \ a \ \cap \text{under } r \ b) & \\
\text{from } \text{conversion-to-locale}[\text{OF this}] & \\
\text{obtain } \text{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} & \\
\text{ultimately have } \exists \text{xs. lcliff } r \ a \ b \ \text{xs} \land (t, u) \in \text{lconv xs} & \\
\text{by } (\text{intro exI[of } - \text{@ } b' @ \ \text{cs} @ a' @ \text{bs], unfold lconv-append lcliff-def ] blast} & \\
\end{align*}
\}

moreover \{ \\
\begin{align*}
\text{fix } w & \\
\text{assume } (t, w) \in \text{conversion}' (\text{under } r \ a \cup \text{under } r \ b) & 
\end{align*}
\}

25
from conversion-to-lconv[OF this]
obtain cs where snd t 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 t 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 exf[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 t 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 t 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 t 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 exf[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 t 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, y) ∈ conversion' (under r a ∪ under r b)
  from conversion-to-lconv[OF this]
obtain cs where snd t 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 exf[of - as @ b' @ cs], unfold lconv-append rcliff-def) blast
}
ultimately show ?case using rc[OF rcliff] by blast
qed fact+
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`:
  - `\( \forall i. \text{sym } (E i) \)`
- **And** `pk`:
  - `\( \forall a b s t u. (s, t) \in L a \Rightarrow (s, u) \in L b \Rightarrow \)`
  - `\( (t, u) \in \text{conversion'} L L E (\text{under } r a) O (L b) = O \text{ conversion'} L L E (\text{under } r a \cup \text{under } r b) O \)`
  - `\( (L a)^{-1} = O \text{ conversion'} L L E (\text{under } r b) \)`
- **And** `cl`:
  - `\( \forall a b s t u. (s, t) \in L a \Rightarrow (s, u) \in E b \Rightarrow \)`
  - `\( (t, u) \in \text{conversion'} L L E (\text{under } r a) O E b O \text{ conversion'} L L E (\text{under } r a) O \)`
  - `\( (L a)^{-1} = O \text{ conversion'} L L E (\text{under } r a \cap \text{under } r b) \cup \)`
  - `\( (t, u) \in \text{conversion'} L L E (\text{under } r a \cup \text{under } r b) O ((L a)^{-1}) = O \text{ conversion'} L L E (\text{under } r b) \)`

**Shows** `conversion' L L E UNIV \subseteq 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 a,symmetric]]`
  - **By** (auto simp only: `ac-simps E' converse-inward")

**Qed**

### 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`:
  - `\( \forall a b s t u. (s, t) \in L a \Rightarrow (s, u) \in R b \Rightarrow \)`
  - `\( (t, u) \in \text{conversion"} L R \text{ (under } r a \cup \text{under } r b) O \)`
  - `\( (L a)^{-1} = O \text{ conversion"} L R \text{ (under } r b) \)`

**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 \text{ UNIV by regexp}`
- **Also have** `\( \subseteq \text{valley"} L R \text{ UNIV}`
  - **Using** `dd-commute-modulo[OF assms(1,2), of L R \lambda-. \{\} ] 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]]`

27
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 ⊆ r
begin

private abbreviation (input) down :: ('b ⇒ 'a rel) ⇒ ('b ⇒ 'a rel) where
down L ≡ λi. ∪j ∈ under q i. L j

private lemma Union-down: (∪ i. down L i) = (∪ 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 transq reflq compat peak]:
  assumes pk: ∀a b s t u. (s, t) ∈ L a ⇒ (s, u) ∈ R b ⇒
  (t, u) ∈ conversion'' L R (under r a) O (down R b)'' O conversion'' L R (under r a ∪ 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 set-mp[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 transq reflq compat peak] =
edd-commute[of L L for L, unfolded CR-iff-self-commute[symmetric]]
end
References

