# Verification of Query Optimization Algorithms

Bernhard Stöckl

February 6, 2026

**Abstract**

This formalization includes a general framework for query optimization consisting of the definitions of selectivities, query graphs, join trees, and cost functions. Furthermore, it implements the join ordering algorithm IKKBZ using these definitions. It verifies the correctness of these definitions and proves that IKKBZ produces an optimal solution within a restricted solution space.

## Contents

**1 Selectivities** **3**
- 1.1 Selectivity Functions . . . 3
- 1.2 Proofs . . . 4

**2 Join Tree** **12**
- 2.1 Functions . . . 13
  - 2.1.1 Functions for Information Retrieval . . . 13
  - 2.1.2 Functions for Correctness Checks . . . 13
  - 2.1.3 Functions for Modifications . . . 14
  - 2.1.4 Additional properties . . . 14
  - 2.1.5 Cardinality Calculations for Left-deep Trees . . . 14
- 2.2 Proofs . . . 15

**3 Cost Functions** **29**
- 3.1 General Cost Functions . . . 29
- 3.2 Cost functions that are considered by IKKBZ. . . . 30
- 3.3 Properties of Cost Functions . . . 30
- 3.4 Proofs . . . 31
  - 3.4.1 Equivalence Proofs . . . 31
  - 3.4.2 Additional ASI Proofs . . . 52

**4 Graph Extensions** **55**
- 4.1 Vertices with Multiple Outgoing Arcs . . . 61
- 4.2 Vertices with Multiple Incoming Arcs . . . 67

**5 Query Graphs** **68**
5.1 Function for Join Trees and Selectivities . . . 69
5.2 Proofs . . . 69
5.3 Pair Query Graph . . . 76

**6 Directed Tree Additions** **77**
6.1 Directed Trees of Connected Trees . . . 78
6.1.1 Tranformation using BFS . . . 78
6.1.2 Tranformation using PSP-Trees . . . 86
6.2 Additions for Induction on Directed Trees . . . 96
6.3 Branching Points in Directed Trees . . . 102
6.4 Converting to Trees of Lists . . . 106

**7 Algebraic Type for Directed Trees** **108**
7.1 Termination Proofs . . . 108
7.2 Dtree Basic Functions . . . 109
7.3 Dtree Basic Proofs . . . 110
7.3.1 Finite Directed Trees to Dtree . . . 138
7.3.2 Well-Formed Dtrees . . . 151
7.3.3 Identity of Transformation Operations . . . 162
7.4 Degrees of Nodes . . . 168
7.5 List Conversions . . . 178
7.6 Inserting in Dtrees . . . 188

**8 Dtrees of Lists** **209**
8.1 Functions . . . 209
8.2 List Dtrees as Well-Formed Dtrees . . . 210
8.3 Combining Preserves Well-Formedness . . . 219

**9 IKKBZ** **232**
9.1 Additional Proofs for Merging Lists . . . 232
9.2 Merging Subtrees of Ranked Dtrees . . . 236
9.2.1 Definitions . . . 236
9.2.2 Commutativity Proofs . . . 237
9.2.3 Merging Preserves Arcs and Verts . . . 243
9.2.4 Merging Preserves Well-Formedness . . . 255
9.2.5 Additional Merging Properties . . . 259
9.3 Normalizing Dtrees . . . 266
9.3.1 Definitions . . . 266
9.3.2 Basic Proofs . . . 266
9.3.3 Normalizing Preserves Well-Formedness . . . 267
9.3.4 Distinctness and hd preserved . . . 271
9.3.5 Normalize and Sorting . . . 272
9.4 Removing Wedges . . . 278

9.5 IKKBZ-Sub . . . . . 286
9.6 Full IKKBZ . . . . . 294

**10 Optimality of IKKBZ** **298**
10.1 Sublist Additions . . . . . 314
10.2 Optimal Solution for Lists of Fixed Sets . . . . . 320
10.3 Arc Invariants . . . . . 357
10.3.1 Normalizing preserves Arc Invariants . . . . . 374
10.3.2 Merging preserves Arc Invariants . . . . . 398
10.3.3 Merge1 preserves Arc Invariants . . . . . 403
10.4 Optimality of IKKBZ-Sub result constrained to Invariants . . 413
10.4.1 Result fulfills the requirements . . . . . 413
10.4.2 Minimal Cost of the result . . . . . 425
10.5 Arc Invariants hold for Conversion to Dtree . . . . . 435
10.6 Optimality of IKKBZ-Sub . . . . . 439
10.7 Optimality of IKKBZ . . . . . 439

**11 Examples of Applying IKKBZ** **440**
11.1 Computing Contributing Selectivity without Lists . . . . . 440
11.2 Contributing Selectivity Satisfies ASI Property . . . . . 445
11.3 Applying IKKBZ . . . . . 448
11.3.1 Applying IKKBZ on Simple Cost Functions . . . . . 449
11.3.2 Applying IKKBZ on C_out . . . . . 451
11.4 Instantiating Comparators with Linorders . . . . . 453

**theory** *Selectivities*
**imports** *Complex-Main HOL−Library.Multiset*
**begin**

# 1 Selectivities

**type-synonym** *'a selectivity* = *'a* ⇒ *'a* ⇒ *real*

**definition** *sel-symm* :: *'a selectivity* ⇒ *bool* **where**
*sel-symm sel* = (∀ *x y*. *sel x y* = *sel y x*)

**definition** *sel-reasonable* :: *'a selectivity* ⇒ *bool* **where**
*sel-reasonable sel* = (∀ *x y*. *sel x y* ≤ *1* ∧ *sel x y* > *0*)

## 1.1 Selectivity Functions

**fun** *list-sel-aux* :: *'a selectivity* ⇒ *'a* ⇒ *'a list* ⇒ *real* **where**
*list-sel-aux sel x* [] = *1*
| *list-sel-aux sel x* (*y*#*ys*) = *sel x y* ∗ *list-sel-aux sel x ys*

**fun** *list-sel* :: *'a selectivity* ⇒ *'a list* ⇒ *'a list* ⇒ *real* **where**
  *list-sel sel* [] *y* = *1*
| *list-sel sel* (*x*#*xs*) *y* = *list-sel-aux sel x y* ∗ *list-sel sel xs y*

**fun** *list-sel-aux'* :: *'a selectivity* ⇒ *'a list* ⇒ *'a* ⇒ *real* **where**
  *list-sel-aux' sel* [] *y* = *1*
| *list-sel-aux' sel* (*x*#*xs*) *y* = *sel x y* ∗ *list-sel-aux' sel xs y*

**fun** *list-sel'*:: *'a selectivity* ⇒ *'a list* ⇒ *'a list* ⇒ *real* **where**
  *list-sel' sel x* [] = *1*
| *list-sel' sel x* (*y*#*ys*) = *list-sel-aux' sel x y* ∗ *list-sel' sel x ys*

**definition** *set-sel-aux* :: *'a selectivity* ⇒ *'a* ⇒ *'a set* ⇒ *real* **where**
  *set-sel-aux sel x Y* = (∏ *y* ∈ *Y*. *sel x y*)

**definition** *set-sel* :: *'a selectivity* ⇒ *'a set* ⇒ *'a set* ⇒ *real* **where**
  *set-sel sel X Y* = (∏ *x* ∈ *X*. *set-sel-aux sel x Y*)

**definition** *set-sel-aux'* :: *'a selectivity* ⇒ *'a set* ⇒ *'a* ⇒ *real* **where**
  *set-sel-aux' sel X y* = (∏ *x* ∈ *X*. *sel x y*)

**definition** *set-sel'* :: *'a selectivity* ⇒ *'a set* ⇒ *'a set* ⇒ *real* **where**
  *set-sel' sel X Y* = (∏ *y* ∈ *Y*. *set-sel-aux' sel X y*)

**fun** *ldeep-s* :: *'a selectivity* ⇒ *'a list* ⇒ *'a* ⇒ *real* **where**
  *ldeep-s f* [] = (λ-. *1*)
| *ldeep-s f* (*x*#*xs*) = (λ*a*. *if a*=*x then list-sel-aux' f xs a else ldeep-s f xs a*)

## 1.2 Proofs

**lemma** *distinct-alt*: (∀ *x*∈# *mset xs*. *count* (*mset xs*) *x* = *1*) ⟷ *distinct xs*
  **by**(*induction xs*) *auto*

**lemma** *mset-y-eq-list-sel-aux-eq*: *mset y* = *mset z* ⟹ *list-sel-aux f x y* = *list-sel-aux f x z*
**proof**(*induction length y arbitrary*: *y z*)
  **case** *0*
  **then show** *?case* **by** *simp*
**next**
  **case** (*Suc n*)
  **then have** *length y* > *0* **by** *auto*
  **then obtain** *y' ys* **where** *y-def*[*simp*]: *y*=*y'*#*ys* **using** *list.exhaust-sel* **by** *blast*
  **have** *length z* > *0* **using** *Suc* **by** *auto*
  **then obtain** *z' zs* **where** *z-def*[*simp*]: *z*=*z'*#*zs* **using** *list.exhaust-sel* **by** *blast*
 **then have** *length zs* = *n* **using** *Suc* **by** (*metis length-Cons mset-eq-length nat.inject*)
  **then show** *?case*
  **proof**(*cases y'*=*z'*)
    **case** *True*
    **then show** *?thesis* **using** *Suc* **by** *simp*

```
  next
    case False
    have y' ∈# mset y by simp
    moreover have z' ∈# mset y using Suc by simp
    ultimately have ∃ c. mset y = mset (y'#z'#c)
      using False multi-member-split [of z' ‹mset ys›]
      by auto (metis ex-mset)
    then obtain c where c-def[simp]: mset y = mset (y'#z'#c) by blast
    then have 0: mset ys = mset (z'#c) by simp
    then have 1: mset zs = mset (y'#c) using Suc.prems by simp
    have list-sel-aux f x y = list-sel-aux f x (y' # ys) by simp
    also have ... = f x y' * list-sel-aux f x ys by simp
    also have ... = f x y' * list-sel-aux f x (z'#c) using Suc.hyps 0 by fastforce
    also have ... = f x z' * list-sel-aux f x (y'#c) by simp
    also have ... = f x z' * list-sel-aux f x zs
      using 1 Suc.hyps(1) ‹length zs = n› by presburger
    finally show ?thesis by simp
  qed
qed

lemma mset-y-eq-list-sel-eq: mset y = mset y' ⟹ list-sel f x y = list-sel f x y'
  apply(induction x)
   apply(auto)[2]
  using mset-y-eq-list-sel-aux-eq by fast

lemma mset-x-eq-list-sel-eq: mset x = mset z ⟹ list-sel f x y = list-sel f z y
proof(induction length x arbitrary: x z)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then have length x > 0 by auto
  then obtain x' xs where y-def[simp]: x=x'#xs using list.exhaust-sel by blast
  have length z > 0 using Suc by auto
  then obtain z' zs where z-def[simp]: z=z'#zs using list.exhaust-sel by blast
 then have length zs = n using Suc by (metis length-Cons mset-eq-length nat.inject)
  then show ?case
  proof(cases x'=z')
    case True
    then show ?thesis using Suc by simp
  next
    case False
    have x' ∈# mset x by simp
    moreover have z' ∈# mset x using Suc by simp
    ultimately have ∃ c. mset x = mset (x'#z'#c)
      using False multi-member-split [of z' ‹mset xs›]
      by auto (metis ex-mset)
    then obtain c where c-def[simp]: mset x = mset (x'#z'#c) by blast
    then have 0: mset xs = mset (z'#c) by simp
```

```
    then have 1: mset zs = mset (x'#c) using Suc.prems by simp
    have list-sel f x y = list-sel f (x'#xs) y by simp
    also have ... = list-sel-aux f x' y * list-sel f xs y by simp
     also have ... = list-sel-aux f x' y * list-sel f (z'#c) y using Suc.hyps 0 by
fastforce
    also have ... = list-sel-aux f z' y * list-sel f (x'#c) y by simp
    also have ... = list-sel-aux f z' y * list-sel f zs y
      using 1 Suc.hyps(1) ‹length zs = n› by presburger
    finally show ?thesis by simp
  qed
qed

lemma list-sel-empty: list-sel f x [] = 1
  by(induction x) auto

lemma list-sel'-empty: list-sel' f [] y = 1
  by(induction y) auto

lemma list-sel-symm-app:
    sel-symm f ⟹ list-sel-aux f x y * list-sel f y xs = list-sel f y (x # xs)
  by(induction y) (auto simp: sel-symm-def)

lemma list-sel-symm: sel-symm f ⟹ list-sel f x y = list-sel f y x
  by(induction x) (auto simp: sel-symm-def list-sel-empty list-sel-symm-app)

lemma list-sel-symm-aux-eq': sel-symm f ⟹ list-sel-aux f x y = list-sel-aux' f y
x
  by(induction y) (auto simp: sel-symm-def)

lemma list-sel-sing-aux': list-sel f x [y] = list-sel-aux' f x y
  by(induction x) auto

lemma list-sel-sing-aux: list-sel f [x] y = list-sel-aux f x y
  by(induction y) auto

lemma list-sel'-sing-aux': list-sel' f x [y] = list-sel-aux' f x y
  by(induction x) auto

lemma list-sel'-sing-aux: list-sel' f [x] y = list-sel-aux f x y
  by(induction y) auto

lemma list-sel'-split-aux: list-sel' f (x#xs) y = list-sel-aux f x y * list-sel' f xs y
  by(induction y) auto

lemma list-sel-eq': list-sel f x y = list-sel' f x y
  by(induction x) (auto simp: list-sel'-empty list-sel'-split-aux)

lemma mset-x-eq-list-sel-aux'-eq: mset x = mset z ⟹ list-sel-aux' f x y = list-sel-aux'
f z y
```

**using** *list-sel-sing-aux'* *mset-x-eq-list-sel-eq* **by** *metis*

**lemma** *foldl-acc-extr*: *foldl* (λ*a b*. *a* ∗ *f x b*) *z y* = *z* ∗ *foldl* (λ*a b*. *a* ∗ *f x b*) (*1*::*real*) *y*
**proof**(*induction y arbitrary*: *z*)
**case** *Nil*
**then show** *?case* **by** *simp*
**next**
**case** (*Cons y ys*)
**have** *foldl* (λ*a b*. *a* ∗ *f x b*) *z* (*y* # *ys*) = *foldl* (λ*a b*. *a* ∗ *f x b*) (*z* ∗ *f x y*) *ys* **by** *simp*
**also have** ... = (*z* ∗ *f x y*) ∗ *foldl* (λ*a b*. *a* ∗ *f x b*) *1 ys* **using** *Cons* **by** *blast*
**also have** ... = *z* ∗ *foldl* (λ*a b*. *a* ∗ *f x b*) *1* (*y*#*ys*)
**by** (*smt* (*verit*, *ccfv-SIG*) *Cons.IH foldl-Cons mult.assoc mult.left-commute*)
**finally show** *?case* **.**
**qed**

**lemma** *list-sel-aux-eq-foldl*: *list-sel-aux f x y* = *foldl* (λ*a b*. *a* ∗ *f x b*) *1 y*
**apply**(*induction y*)
**apply**(*auto*)[*2*]
**using** *foldl-acc-extr* **by** *metis*

**lemma** *list-sel-eq-foldl*: *list-sel f x y* = *foldl* (λ*a b*. *a* ∗ *list-sel-aux f b y*) *1 x*
**apply**(*induction x*)
**apply**(*auto*)[*2*]
**using** *foldl-acc-extr* **by** *metis*

**corollary** *list-sel-eq-foldl2*: *list-sel f x y* = *foldl* (λ*a x*. *a* ∗ *foldl* (λ*a b*. *a* ∗ *f x b*) *1 y*) *1 x*
**by** (*simp add*: *list-sel-aux-eq-foldl list-sel-eq-foldl*)

**lemma** *list-sel-aux-eq-foldr*: *list-sel-aux f x y* = *foldr* (λ*b a*. *a* ∗ *f x b*) *y 1*
**by**(*induction y*) *auto*

**lemma** *sel-foldl-eq-foldr*:
*foldl* (λ*a b*. *a* ∗ *f x b*) *1 y* = *foldr* (λ*b a*. *a* ∗ (*f*::*'a selectivity*) *x b*) *y 1*
**using** *list-sel-aux-eq-foldl list-sel-aux-eq-foldr* **by** *metis*

**lemma** *list-sel-eq-foldr*: *list-sel f x y* = *foldr* (λ*b a*. *a* ∗ *list-sel-aux f b y*) *x 1*
**by**(*induction x*) *auto*

**lemma** *list-sel-eq-foldr2*: *list-sel f x y* = *foldr* (λ*x a*. *a* ∗ *foldr* (λ*b a*. *a* ∗ *f x b*) *y 1*) *x 1*
**by** (*simp add*: *list-sel-aux-eq-foldr list-sel-eq-foldr*)

**lemma** *list-sel-aux-reasonable*:
*sel-reasonable f* ⟹ *list-sel-aux f x y* ≤ *1* ∧ *list-sel-aux f x y* > *0*
**by**(*induction y*) (*auto simp*: *sel-reasonable-def mult-le-one*)

**lemma** *list-sel-aux'-reasonable*:
  *sel-reasonable f* $\Longrightarrow$ *list-sel-aux' f x y* $\leq$ *1* $\wedge$ *list-sel-aux' f x y* $>$ *0*
 **by**(*induction x*) (*auto simp*: *sel-reasonable-def mult-le-one*)

**lemma** *list-sel-reasonable*: *sel-reasonable f* $\Longrightarrow$ *list-sel f x y* $\leq$ *1* $\wedge$ *list-sel f x y* $>$ *0*
 **by**(*induction x*) (*auto simp*: *sel-reasonable-def mult-le-one list-sel-aux-reasonable*)

**lemma** *list-sel'-reasonable*: *sel-reasonable f* $\Longrightarrow$ *list-sel' f x y* $\leq$ *1* $\wedge$ *list-sel' f x y* $>$ *0*
 **using** *list-sel-eq' list-sel-reasonable* **by** *metis*

**lemma** *list-sel-aux-eq-set-sel-aux*:
 *distinct ys* $\Longrightarrow$ *list-sel-aux f x ys* $=$ *set-sel-aux f x* (*set ys*)
 **by**(*induction ys*) (*auto simp*: *set-sel-aux-def*)

**lemma** *list-sel-eq-set-sel*:
 $[\![$*distinct xs*; *distinct ys*$]\!] \Longrightarrow$ *list-sel f xs ys* $=$ *set-sel f* (*set xs*) (*set ys*)
 **by**(*induction xs*) (*auto simp*: *set-sel-def list-sel-aux-eq-set-sel-aux list-sel-empty*)

**lemma** *list-sel'-eq-set-sel*:
 $[\![$*distinct xs*; *distinct ys*$]\!] \Longrightarrow$ *list-sel' f xs ys* $=$ *set-sel f* (*set xs*) (*set ys*)
 **by** (*auto simp add*: *list-sel-eq' dest*: *list-sel-eq-set-sel*)

**lemma** *set-sel-symm-if-finite*: $[\![$*finite X*; *finite Y*; *sel-symm f*$]\!] \Longrightarrow$ *set-sel f X Y* $=$ *set-sel f Y X*
 **using** *finite-distinct-list list-sel-symm list-sel-eq-set-sel* **by** *metis*

**lemma** *set-sel-aux-1-if-notfin*: $\neg$*finite Y* $\Longrightarrow$ *set-sel-aux f x Y* $=$ *1*
 **unfolding** *set-sel-aux-def* **by** *simp*

**lemma** *set-sel-1-if-notfin1*: $\neg$*finite X* $\Longrightarrow$ *set-sel f X Y* $=$ *1*
 **unfolding** *set-sel-def set-sel-aux-def* **by** *simp*

**lemma** *set-sel-1-if-notfin2*: $\neg$*finite Y* $\Longrightarrow$ *set-sel f X Y* $=$ *1*
 **unfolding** *set-sel-def set-sel-aux-def* **by** *simp*

**lemma** *set-sel-symm*: *sel-symm f* $\Longrightarrow$ *set-sel f X Y* $=$ *set-sel f Y X*
 **using** *set-sel-symm-if-finite*[*of X Y*]
 **by** (*fastforce simp*: *set-sel-1-if-notfin1 set-sel-1-if-notfin2*)

**lemma** *list-sel-aux'-eq-set-sel-aux'*:
 *distinct xs* $\Longrightarrow$ *list-sel-aux' f xs x* $=$ *set-sel-aux' f* (*set xs*) *x*
 **by**(*induction xs*) (*auto simp*: *set-sel-aux'-def*)

**lemma** *list-sel'-eq-set-sel'*:
 $[\![$*distinct xs*; *distinct ys*$]\!] \Longrightarrow$ *list-sel' f xs ys* $=$ *set-sel' f* (*set xs*) (*set ys*)
 **by**(*induction ys*) (*auto simp*: *set-sel'-def list-sel-aux'-eq-set-sel-aux' list-sel-empty*)

**lemma** *list-sel-eq-set-sel'*:
⟦*distinct xs*; *distinct ys*⟧ ⟹ *list-sel f xs ys* = *set-sel' f* (*set xs*) (*set ys*)
**by** (*simp add*: *list-sel'-eq-set-sel' list-sel-eq'*)

**lemma** *set-sel'-symm-if-finite*: ⟦*finite X*; *finite Y*; *sel-symm f*⟧ ⟹ *set-sel' f X Y* = *set-sel' f Y X*
**using** *finite-distinct-list list-sel-symm list-sel-eq-set-sel'* **by** *metis*

**lemma** *set-sel-aux'-1-if-notfin*: ¬*finite X* ⟹ *set-sel-aux' f X y* = *1*
**unfolding** *set-sel-aux'-def* **by** *simp*

**lemma** *set-sel'-1-if-notfin1*: ¬*finite X* ⟹ *set-sel' f X Y* = *1*
**unfolding** *set-sel'-def set-sel-aux'-def* **by** *simp*

**lemma** *set-sel'-1-if-notfin2*: ¬*finite Y* ⟹ *set-sel' f X Y* = *1*
**unfolding** *set-sel'-def set-sel-aux'-def* **by** *simp*

**lemma** *set-sel'-symm*: *sel-symm f* ⟹ *set-sel' f X Y* = *set-sel' f Y X*
**using** *set-sel'-symm-if-finite*[*of X Y*]
**by** (*fastforce simp*: *set-sel'-1-if-notfin1 set-sel'-1-if-notfin2*)

**lemma** *set-sel'-eq-set-sel*: *set-sel' f X Y* = *set-sel f X Y*
**unfolding** *set-sel-def set-sel-aux-def set-sel'-def set-sel-aux'-def* **using** *prod.swap*
**by** *fast*

**lemma** *set-sel-aux-reasonable-fin*:
⟦*finite y*; *sel-reasonable f*⟧ ⟹ *set-sel-aux f x y* ≤ *1* ∧ *set-sel-aux f x y* > *0*
**unfolding** *set-sel-aux-def*
**by**(*induction y rule*: *finite-induct*) (*auto simp*: *sel-reasonable-def mult-le-one*)

**lemma** *set-sel-aux-reasonable*:
*sel-reasonable f* ⟹ *set-sel-aux f x y* ≤ *1* ∧ *set-sel-aux f x y* > *0*
**by**(*cases finite y*) (*auto simp*: *set-sel-aux-reasonable-fin set-sel-aux-1-if-notfin*)

**lemma** *set-sel-aux'-reasonable-fin*:
⟦*finite x*; *sel-reasonable f*⟧ ⟹ *set-sel-aux' f x y* ≤ *1* ∧ *set-sel-aux' f x y* > *0*
**unfolding** *set-sel-aux'-def*
**by**(*induction x rule*: *finite-induct*) (*auto simp*: *sel-reasonable-def mult-le-one*)

**lemma** *set-sel-aux'-reasonable*:
*sel-reasonable f* ⟹ *set-sel-aux' f x y* ≤ *1* ∧ *set-sel-aux' f x y* > *0*
**by**(*cases finite x*) (*auto simp*: *set-sel-aux'-reasonable-fin set-sel-aux'-1-if-notfin*)

**lemma** *set-sel-reasonable-fin*:
⟦*finite x*; *sel-reasonable f*⟧ ⟹ *set-sel f x y* ≤ *1* ∧ *set-sel f x y* > *0*
**unfolding** *set-sel-def*
**apply**(*induction x rule*: *finite-induct*)
**using** *set-sel-aux'-reasonable-fin* **apply**(*simp*)
**by** (*smt* (*verit*) *prod-le-1 prod-pos set-sel-aux-reasonable*)

**lemma** *set-sel-reasonable*: *sel-reasonable f* $\Longrightarrow$ *set-sel f x y* $\leq$ *1* $\wedge$ *set-sel f x y* $>$ *0*
**by**(*cases finite x*) (*auto simp*: *set-sel-reasonable-fin set-sel-1-if-notfin1*)

**lemma** *set-sel′-reasonable-fin*:
⟦*finite y*; *sel-reasonable f*⟧ $\Longrightarrow$ *set-sel′ f x y* $\leq$ *1* $\wedge$ *set-sel′ f x y* $>$ *0*
**unfolding** *set-sel′-def*
**apply**(*induction y rule*: *finite-induct*)
**using** *set-sel-aux′-reasonable-fin* **apply**(*simp*)
**by** (*smt* (*verit*) *prod-le-1 prod-pos set-sel-aux′-reasonable*)

**lemma** *set-sel′-reasonable*: *sel-reasonable f* $\Longrightarrow$ *set-sel′ f x y* $\leq$ *1* $\wedge$ *set-sel′ f x y* $>$ *0*
**by** (*cases finite y*) (*auto simp*: *set-sel′-reasonable-fin set-sel′-1-if-notfin2*)

**lemma** *ldeep-s-pos*: *sel-reasonable f* $\Longrightarrow$ *ldeep-s f xs x* $>$ *0*
**by** (*induction xs*) (*auto simp*: *list-sel-aux′-reasonable*)

**lemma** *distinct-app-trans-r*: *distinct* (*ys*@*xs*) $\Longrightarrow$ *distinct xs*
**by** *simp*

**lemma** *distinct-app-trans-l*: *distinct* (*ys*@*xs*) $\Longrightarrow$ *distinct ys*
**by** *simp*

**lemma** *ldeep-s-reasonable*: *sel-reasonable f* $\Longrightarrow$ *ldeep-s f xs y* $\leq$ *1* $\wedge$ *ldeep-s f xs y* $>$ *0*
**by** (*induction xs*) (*auto simp*: *list-sel-aux′-reasonable*)

**lemma** *ldeep-s-eq-list-sel-aux′-split*:
*y* $\in$ *set xs* $\Longrightarrow$ $\exists$ *as bs*. *as* @ *y* # *bs* = *xs* $\wedge$ *ldeep-s sel xs y* = *list-sel-aux′ sel bs y*
**proof**(*induction xs*)
**case** (*Cons x xs*)
**then show** *?case*
**proof**(*cases x* = *y*)
**case** *False*
**then obtain** *as bs* **where** *as-def*: *as* @ *y* # *bs* = *xs ldeep-s sel xs y* = *list-sel-aux′ sel bs y*
**using** *Cons* **by** *auto*
**then have** (*x*#*as*) @ *y* # *bs* = *x*#*xs* **by** *simp*
**then show** *?thesis* **using** *False as-def*(*2*) **by** *fastforce*
**qed**(*auto*)
**qed**(*simp*)

**lemma** *distinct-ldeep-s-eq-aux*:
*distinct xs* $\Longrightarrow$ $\exists$ *xs′*. *xs′*@*y*#*ys*=*xs* $\Longrightarrow$ *ldeep-s f xs y* = *list-sel-aux′ f ys y*
**proof**(*induction xs arbitrary*: *ys*)
**case** (*Cons x xs*)
**then show** *?case*

```
  proof(cases x=y ∧ ys=xs)
    case True
    then show ?thesis using Cons.prems by simp
  next
    case False
    then have ∃ xs'. xs'@y#ys=x#xs ∧ xs' ≠ [] using Cons.prems by auto
    then have 0: ∃ xs''. x#xs''@y#ys=x#xs by (metis list.sel(3) tl-append2)
    have 1: distinct xs using Cons.prems(1) by fastforce
    then show ?thesis
    proof(cases x=y)
      case True
      then have count (mset (x#xs)) x ≥ 2 using 0 by auto
      then show ?thesis using Cons.prems by simp
    next
      case False
      then have ldeep-s f (x # xs) y
            = (λa. if a=x then list-sel-aux' f xs a else ldeep-s f xs a) y by simp
      also have ... =  ldeep-s f xs y using False by simp
      finally show ?thesis using Cons.IH 0 1 by simp
    qed
  qed
qed(simp)

lemma distinct-ldeep-s-eq-aux':
  ⟦distinct xs; as @ y # bs = xs⟧ ⟹ ldeep-s sel xs y = list-sel-aux' sel bs y
  using distinct-ldeep-s-eq-aux by fast

lemma ldeep-s-last1-if-distinct: distinct xs ⟹ ldeep-s sel xs (last xs) = 1
  by (induction xs) auto

lemma ldeep-s-revhd1-if-distinct: distinct xs ⟹ ldeep-s sel (rev xs) (hd xs) = 1
  using ldeep-s-last1-if-distinct[of rev xs] by (simp add: last-rev)

lemma ldeep-s-1-if-nelem: x ∉ set xs ⟹ ldeep-s sel xs x = 1
  by (induction xs) auto

lemma distinct-xs-not-ys: distinct (xs@ys) ⟹ x ∈ set xs ⟹ x ∉ set ys
  by auto

lemma distinct-ys-not-xs: distinct (xs@ys) ⟹ x ∈ set ys ⟹ x ∉ set xs
  by auto

lemma distinct-change-order-first-eq-nempty:
  assumes distinct (xs@ys@zs@rs)
      and ys ≠ []
      and zs ≠ []
      and take 1 (xs@ys@zs@rs) = take 1 (xs@zs@ys@rs)
    shows xs ≠ []
proof
```

**assume** *xs* = []
**then have** *take 1* (*ys*@*zs*@*rs*) = *take 1* (*zs*@*ys*@*rs*) **using** *assms*(*4*) **by** *simp*
**then have** ∃ *r rs1 rs2*. *ys*@*zs*@*rs* = *r*#*rs1* ∧ *zs*@*ys*@*rs* = *r*#*rs2*
**by** (*metis append-Cons append-take-drop-id assms*(*3*) *neq-Nil-conv take-eq-Nil zero-neq-one*)
**then obtain** *r rs1 rs2* **where** *r-def*: *ys*@*zs*@*rs* = *r*#*rs1* ∧ *zs*@*ys*@*rs* = *r*#*rs2* **by** *blast*
**then have** *0*: *r* ∈ *set ys* ∧ *r* ∈ *set zs*
**using** *assms*(*2*,*3*) **by** (*metis Cons-eq-append-conv list.set-intros*(*1*))
**then show** *False* **using** *0 assms*(*1*) **by** *auto*
**qed**

**lemma** *distinct-change-order-first-elem*:
⟦*distinct* (*xs*@*ys*@*zs*@*rs*); *ys* ≠ []; *zs* ≠ []; *take 1* (*xs*@*ys*@*zs*@*rs*) = *take 1* (*xs*@*zs*@*ys*@*rs*)⟧
⟹ *take 1* (*xs*@*ys*@*zs*@*rs*) = *take 1 xs*
**by** (*cases xs*) (*fastforce dest*!: *distinct-change-order-first-eq-nempty*)+

**lemma** *take1-singleton-app*: *take 1 xs* = [*r*] ⟹ *take 1* (*xs*@*ys*) = [*r*]
**by** (*induction xs*) (*auto*)

**lemma** *hd-eq-take1*: *take 1 xs* = [*r*] ⟹ *hd xs* = *r*
**using** *hd-take*[*of 1 xs*] **by** *simp*

**lemma** *take1-eq-hd*: ⟦*xs* ≠ []; *hd xs* = *r*⟧ ⟹ *take 1 xs* = [*r*]
**by** (*simp add*: *take-Suc*)

**lemma** *nempty-if-take1*: *take 1 xs* = [*r*] ⟹ *xs* ≠ []
**by** *force*

**end**

**theory** *JoinTree*
**imports** *Complex-Main HOL−Library.Multiset Selectivities*
**begin**

# 2 Join Tree

Relations have an identifier and cardinalities. Joins have two children and a result cardinality. The datatype only represents the structure while cardinalities are given by a separate function.

**datatype** (*relations*:'*a*) *joinTree* = *Relation* '*a* | *Join* '*a joinTree* '*a joinTree*

**type-synonym** '*a card* = '*a* ⇒ *real*

## 2.1 Functions

### 2.1.1 Functions for Information Retrieval

**fun** *inorder* :: *'a joinTree* ⇒ *'a list* **where**
  *inorder* (*Relation rel*) = [*rel*]
| *inorder* (*Join l r*) = *inorder l* @ *inorder r*

**fun** *revorder* :: *'a joinTree* ⇒ *'a list* **where**
  *revorder* (*Relation rel*) = [*rel*]
| *revorder* (*Join l r*) = *revorder r* @ *revorder l*

**fun** *relations-mset* :: *'a joinTree* ⇒ *'a multiset* **where**
  *relations-mset* (*Relation rel*) = {#*rel*#}
| *relations-mset* (*Join l r*) = *relations-mset l* + *relations-mset r*

**fun** *card* :: *'a card* ⇒ *'a selectivity* ⇒ *'a joinTree* ⇒ *real* **where**
  *card cf f* (*Relation rel*) = *cf rel*
| *card cf f* (*Join l r*) =
    *list-sel f* (*inorder l*) (*inorder r*) ∗ *card cf f l* ∗ *card cf f r*

**fun** *cards-list* :: *'a card* ⇒ *'a joinTree* ⇒ (*'a*×*real*) *list* **where**
  *cards-list cf* (*Relation rel*) = [(*rel*,*cf rel*)]
| *cards-list cf* (*Join l r*) = *cards-list cf l* @ *cards-list cf r*

**fun** *height* :: *'a joinTree* ⇒ *nat* **where**
  *height* (*Relation* -) = *0*
| *height* (*Join l r*) = *max* (*height l*) (*height r*) + *1*

**fun** *num-relations* :: *'a joinTree* ⇒ *nat* **where**
  *num-relations* (*Relation* -) = *1*
| *num-relations* (*Join l r*) = *num-relations l* + *num-relations r*

**fun** *first-node* :: *'a joinTree* ⇒ *'a* **where**
  *first-node* (*Relation r*) = *r*
| *first-node* (*Join l* -) = *first-node l*

### 2.1.2 Functions for Correctness Checks

Cardinalities must be positive and selectivities need to be ∈ (*0*,*1*].

**fun** *reasonable-cards* :: *'a card* ⇒ *'a selectivity* ⇒ *'a joinTree* ⇒ *bool* **where**
  *reasonable-cards cf f* (*Relation rel*) = (*cf rel* > *0*)
| *reasonable-cards cf f* (*Join l r*) = (*let c* = *card cf f* (*Join l r*) *in*
    *c* ≤ *card cf f l* ∗ *card cf f r* ∧ *c* > *0* ∧ *reasonable-cards cf f l* ∧ *reasonable-cards cf f r*)

**definition** *pos-rel-cards* :: *'a card* ⇒ *'a joinTree* ⇒ *bool* **where**
  *pos-rel-cards cf t* = (∀ (-,*c*)∈*set* (*cards-list cf t*). *c* > *0*)

**definition** *pos-list-cards* :: *'a card* ⇒ *'a list* ⇒ *bool* **where**

*pos-list-cards cf xs* = (∀ *x*∈*set xs*. *cf x* > *0*)

Each node should have a unique identifier.

**definition** *distinct-relations* :: *'a joinTree* ⇒ *bool* **where**
*distinct-relations t* = *distinct* (*inorder t*)

### 2.1.3 Functions for Modifications

**fun** *mirror* :: *'a joinTree* ⇒ *'a joinTree* **where**
*mirror* (*Relation rel*) = *Relation rel*
| *mirror* (*Join l r*) = *Join* (*mirror r*) (*mirror l*)

**fun** *create-rdeep* :: *'a list* ⇒ *'a joinTree* **where**
*create-rdeep* [] = *undefined*
| *create-rdeep* [*x*] = *Relation x*
| *create-rdeep* (*x*#*xs*) = *Join* (*Relation x*) (*create-rdeep xs*)

**fun** *create-ldeep-rev* :: *'a list* ⇒ *'a joinTree* **where**
*create-ldeep-rev* [] = *undefined*
| *create-ldeep-rev* [*x*] = *Relation x*
| *create-ldeep-rev* (*x*#*xs*) = *Join* (*create-ldeep-rev xs*) (*Relation x*)

**definition** *create-ldeep* :: *'a list* ⇒ *'a joinTree* **where**
*create-ldeep xs* = *create-ldeep-rev* (*rev xs*)

### 2.1.4 Additional properties

**fun** *left-deep* :: *'a joinTree* ⇒ *bool* **where**
*left-deep* (*Relation* -) = *True*
| *left-deep* (*Join l* (*Relation* -)) = *left-deep l*
| *left-deep* - = *False*

**fun** *right-deep* :: *'a joinTree* ⇒ *bool* **where**
*right-deep* (*Relation* -) = *True*
| *right-deep* (*Join* (*Relation* -) *r*) = *right-deep r*
| *right-deep* - = *False*

**fun** *zig-zag* :: *'a joinTree* ⇒ *bool* **where**
*zig-zag* (*Relation* -) = *True*
| *zig-zag* (*Join l* (*Relation* -)) = *zig-zag l*
| *zig-zag* (*Join* (*Relation* -) *r*) = *zig-zag r*
| *zig-zag* - = *False*

### 2.1.5 Cardinality Calculations for Left-deep Trees

Expects a reversed list of relations rs and calculates the cardinality of a left-deep tree.

**fun** *ldeep-n* :: *'a selectivity* ⇒ *'a card* ⇒ *'a list* ⇒ *real* **where**
*ldeep-n f cf* [] = *1*

| *ldeep-n f cf* (*r*#*rs*) = *cf r* ∗ (*list-sel-aux*′ *f rs r*) ∗ *ldeep-n f cf rs*

**definition** *ldeep-T* :: (′*a* ⇒ *real*) ⇒ ′*a card* ⇒ ′*a list* ⇒ *real* **where**
*ldeep-T sf cf xs* = *foldl* (λ*a b*. *a* ∗ *cf b* ∗ *sf b*) *1 xs*

**fun** *ldeep-T*′ :: (′*a* ⇒ *real*) ⇒ ′*a card* ⇒ ′*a list* ⇒ *real* **where**
*ldeep-T*′ *f cf* [] = *1*
| *ldeep-T*′ *f cf* (*r*#*rs*) = *cf r* ∗ *f r* ∗ *ldeep-T*′ *f cf rs*

## 2.2 Proofs

**lemma** *ldeep-eq-rdeep*: *left-deep t* = *right-deep* (*mirror t*)
**by**(*induction t rule*: *left-deep.induct*) (*auto*)

**lemma** *mirror-twice-id*[*simp*]: *mirror* (*mirror t*) = *t*
**by**(*induction t*) *auto*

**lemma** *rdeep-eq-ldeep*: *right-deep t* = *left-deep* (*mirror t*)
**apply**(*induction t rule*: *right-deep.induct*)
**by**(*auto*)

**lemma** *mirror-zig-zag-preserv*: *zig-zag* (*mirror t*) = *zig-zag t*
**apply**(*induction t rule*: *zig-zag.induct*)
**using** *zig-zag.elims*(*2*) **by** *fastforce*+

**lemma** *ldeep-zig-zag*: *left-deep t* ⟹ *zig-zag t*
**by**(*induction t rule*: *zig-zag.induct*) *auto*

**lemma** *rdeep-zig-zag*: *right-deep t* ⟹ *zig-zag t*
**using** *rdeep-eq-ldeep ldeep-zig-zag mirror-zig-zag-preserv* **by** *blast*

**lemma** *relations-nempty*: *relations t* ≠ {}
**by** (*induction t*) *auto*

**lemma** *set-implies-mset*: *x* ∈ *relations t* ⟹ *x* ∈# *relations-mset t*
**by**(*induction t*) (*auto*)

**lemma** *mset-implies-set*: *x* ∈# *relations-mset t* ⟹ *x* ∈ *relations t*
**by**(*induction t*) (*auto*)

**lemma** *inorder-eq-mset*: *mset* (*inorder t*) = *relations-mset t*
**by**(*induction t*) (*auto*)

**lemma** *relations-set-eq-mset*: *set-mset* (*relations-mset t*) = *relations t*
**using** *mset-implies-set set-implies-mset* **by** *fast*

**lemma** *inorder-eq-set*: *set* (*inorder t*) = *relations t*
**by**(*induction t*) (*auto*)

**lemma** *revorder-eq-mset*: *mset* (*revorder t*) = *relations-mset t*
**by**(*induction t*) (*auto*)

**lemma** *revorder-eq-set*: *set* (*revorder t*) = *relations t*
**by**(*induction t*) (*auto*)

**lemma** *revorder-eq-rev-inorder*: *revorder t* = *rev* (*inorder t*)
**by**(*induction t*) (*auto*)

**lemma** *inorder-eq-rev-revorder*: *inorder t* = *rev* (*revorder t*)
**by**(*induction t*) (*auto*)

**lemma** *mirror-mset-eq*[*simp*]: *relations-mset* (*mirror t*) = *relations-mset t*
**by**(*induction t*) *auto*

**lemma** *distinct-rels-alt*: *distinct-relations t* ⟷ *distinct* (*revorder t*)
**unfolding** *distinct-relations-def inorder-eq-rev-revorder* **by** *simp*

**lemma** *distinct-rels-alt'*:
*distinct-relations t* ⟷ (*let multi*=*relations-mset t in* ∀ *x*∈# *multi*. *count multi x* = *1*)
**using** *distinct-relations-def inorder-eq-mset distinct-alt* **by** *metis*

**lemma** *inorder-nempty*: *inorder t* ≠ []
**by** (*induction t*) *auto*

**lemma** *revorder-nempty*: *revorder t* ≠ []
**by** (*induction t*) *auto*

**lemma** *mirror-distinct*: *distinct-relations t* ⟹ *distinct-relations* (*mirror t*)
**by**(*simp add*: *distinct-rels-alt'*)

**lemma** *mirror-set-eq*[*simp*]: *relations* (*mirror t*) = *relations t*
**by**(*induction t*) *auto*

**lemma** *mirror-inorder-rev*: *inorder* (*mirror t*) = *rev* (*inorder t*)
**by**(*induction t*) *auto*

**lemma** *mirror-revorder-rev*: *revorder* (*mirror t*) = *rev* (*revorder t*)
**by**(*induction t*) *auto*

**corollary** *mirror-revorder-inorder*: *revorder* (*mirror t*) = *inorder t*
**unfolding** *mirror-revorder-rev inorder-eq-rev-revorder* **by** *simp*

**corollary** *mirror-inorder-revorder*: *inorder* (*mirror t*) = *revorder t*
**unfolding** *mirror-inorder-rev revorder-eq-rev-inorder* **by** *simp*

**lemma** *mirror-card-eq*[*simp*]: *sel-symm f* ⟹ *card cf f* (*mirror t*) = *card cf f t*
**proof**(*induction t*)

**case** (*Join l r*)
**let** *?r* = *mirror r* **and** *?l* = *mirror l*
**have** *0*: *mset* (*inorder ?r*) = *mset* (*inorder r*) **by** (*simp add*: *inorder-eq-mset*)
**have** *1*: *mset* (*inorder ?l*) = *mset* (*inorder l*) **by** (*simp add*: *inorder-eq-mset*)
**have** *card cf f* (*mirror* (*Join l r*)) = *card cf f* (*Join* (*mirror r*) (*mirror l*)) **by** *simp*
**also have** ... = *list-sel f* (*inorder ?r*) (*inorder ?l*) ∗ *card cf f r* ∗ *card cf f l*
**using** *Join* **by** *simp*
**also have** ... = *list-sel f* (*inorder r*) (*inorder ?l*) ∗ *card cf f r* ∗ *card cf f l*
**using** *0 mset-x-eq-list-sel-eq* **by** *auto*
**also have** ... = *list-sel f* (*inorder r*) (*inorder l*) ∗ *card cf f r* ∗ *card cf f l*
**using** *1 mset-y-eq-list-sel-eq* **by** *auto*
**finally show** *?case* **using** *list-sel-symm Join.prems* **by** *auto*
**qed**(*simp*)

**lemma** *mirror-reasonable-cards*:
⟦*sel-symm f*; *reasonable-cards cf f t*⟧ ⟹ *reasonable-cards cf f* (*mirror t*)
**proof**(*induction t*)
**case** (*Join l r*)
**let** *?r* = *mirror r* **and** *?l* = *mirror l*
**let** *?c* = *card cf f* (*mirror* (*Join l r*))
**let** *?c′* = *card cf f* (*Join l r*)
**have** *reasonable-cards cf f* (*mirror* (*Join l r*))
= *reasonable-cards cf f* (*Join* (*mirror r*) (*mirror l*)) **by** *simp*
**also have** ... = (*?c* ≤ *card cf f ?r* ∗ *card cf f ?l* ∧ *?c*>*0*
∧ *reasonable-cards cf f ?l* ∧ *reasonable-cards cf f ?r*)
**by** (*auto simp*: *Let-def*)
**also have** ... = (*?c* ≤ *card cf f ?r* ∗ *card cf f ?l* ∧ *?c*>*0*)
**using** *Join* **by** *fastforce*
**also have** ... = (*?c′* ≤ *card cf f r* ∗ *card cf f l* ∧ *?c′*>*0*)
**using** *mirror-card-eq Join.prems* **by** *metis*
**also have** ... = (*?c′* ≤ *card cf f r* ∗ *card cf f l* ∧ *?c′*>*0*
∧ *reasonable-cards cf f l* ∧ *reasonable-cards cf f r*)
**using** *Join.prems* **by** *auto*
**also have** ... = (*?c′* ≤ *card cf f l* ∗ *card cf f r* ∧ *?c′*>*0*
∧ *reasonable-cards cf f l* ∧ *reasonable-cards cf f r*)
**by** *argo*
**finally show** *?case* **using** *Join.prems* **by** *force*
**qed**(*simp*)

**lemma** *joinTree-cases*: (∃ *r*. *t*=(*Relation r*)) ∨ (∃ *l rr*. *t*=(*Join l* (*Relation rr*)))
∨ (∃ *l lr rr*. *t*=(*Join l* (*Join lr rr*)))
**apply**(*cases t*)
**apply**(*auto*)[*2*]
**by** (*meson joinTree.exhaust*)

**lemma** *joinTree-cases-ldeep*: *left-deep t*
⟹ (∃ *r*. *t*=(*Relation r*)) ∨ (∃ *l rr*. *t*=(*Join l* (*Relation rr*)))
**apply**(*cases t*)

**apply**(*auto*)[*2*]
**using** *joinTree-cases* **by** *fastforce*

**lemma** *ldeep-trans*: *left-deep* (*Join l r*) ⟹ *left-deep l*
**by**(*cases r*) *auto*

**lemma** *subtree-elem-count-l*:
**assumes** ∀ *x*∈# (*relations-mset* (*Join l r*)). *count* (*relations-mset* (*Join l r*)) *x* = *1*
**and** *x* ∈# *relations-mset l*
**shows** *count* (*relations-mset l*) *x* = *1*
**proof** −
**have** *0*: *count* (*relations-mset l*) *x* ≥ *1* **using** *assms* **by** *auto*
**have** *count* (*relations-mset l*) *x* ≤ *1* **using** *assms* **by** *force*
**then show** *?thesis* **using** *0* **by** *linarith*
**qed**

**lemma** *subtree-elem-count-r*:
**assumes** ∀ *x*∈# (*relations-mset* (*Join l r*)). *count* (*relations-mset* (*Join l r*)) *x* = *1*
**and** *x* ∈# *relations-mset r*
**shows** *count* (*relations-mset r*) *x* = *1*
**proof** −
**have** *0*: *count* (*relations-mset r*) *x* ≥ *1* **using** *assms* **by** *auto*
**have** *count* (*relations-mset r*) *x* ≤ *1* **using** *assms* **by** *force*
**then show** *?thesis* **using** *0* **by** *linarith*
**qed**

**lemma** *first-node-first-inorder*: ∃ *xs*. *inorder t* = *first-node t* # *xs*
**by**(*induction t*) *auto*

**lemma** *first-node-last-revorder*: ∃ *xs*. *revorder t* = *xs* @ [*first-node t*]
**by**(*induction t*) *auto*

**lemma** *first-node-eq-hd*: *first-node t* = *hd* (*inorder t*)
**using** *first-node-first-inorder*[*of t*] **by** *auto*

**lemma** *distinct-elem-right-not-left*:
**assumes** *distinct-relations* (*Join l r*)
**and** *x* ∈ *relations r*
**shows** *x* ∉ *relations l*
**proof**
**assume** *x* ∈ *relations l*
**then have** *x* ∈# *relations-mset l* **using** *set-implies-mset* **by** *fast*
**then have** *0*: *count* (*relations-mset l*) *x* ≥ *1* **by** *simp*
**have** *x* ∈# *relations-mset r* **using** *set-implies-mset assms*(*2*) **by** *fast*
**then have** *count* (*relations-mset r*) *x* ≥ *1* **by** *simp*
**moreover have** *count* (*relations-mset l* + *relations-mset r*) *x*
= *count* (*relations-mset l*) *x* + *count* (*relations-mset r*) *x* **by** *simp*

**ultimately have** *count* (*relations-mset l* + *relations-mset r*) *x* ≥ *2* **using** *0* **by** *linarith*
**then have** *count* (*relations-mset* (*Join l r*)) *x* ≥ *2* **by** *simp*
**then have** *1*: *count* (*relations-mset* (*Join l r*)) *x* ≠ *1* **by** *simp*
**let** *?multi* = (*relations-mset* (*Join l r*))
**have** *distinct-relations* (*Join l r*) = (∀ *y*∈# *?multi*. *count ?multi y* = *1*)
**by** (*simp add*: *distinct-rels-alt'*)
**then show** *False* **using** *1 assms set-implies-mset* **by** *fastforce*
**qed**

**lemma** *distinct-elem-left-not-right*:
**assumes** *distinct-relations* (*Join l r*)
**and** *x* ∈ *relations l*
**shows** *x* ∉ *relations r*
**using** *distinct-elem-right-not-left assms* **by** *fast*

**lemma** *distinct-relations-disjoint*: *distinct-relations* (*Join l r*) ⟹ *relations l* ∩ *relations r* = {}
**using** *distinct-elem-right-not-left* **by** *fast*

**lemma** *distinct-trans-l*: *distinct-relations* (*Join l r*) ⟹ *distinct-relations l*
**using** *subtree-elem-count-l* **by** (*fastforce simp*: *distinct-rels-alt*)

**lemma** *distinct-trans-r*: *distinct-relations* (*Join l r*) ⟹ *distinct-relations r*
**using** *subtree-elem-count-r* **by** (*fastforce simp*: *distinct-rels-alt*)

**lemma** *distinct-and-disjoint-impl-count1*:
**assumes** *distinct-relations l*
**and** *distinct-relations r*
**and** *relations l* ∩ *relations r* = {}
**and** *x* ∈# *relations-mset* (*Join l r*)
**shows** *count* (*relations-mset* (*Join l r*)) *x* = *1*
**proof** −
**show** *?thesis*
**proof**(*cases x*∈*relations l*)
**case** *True*
**then have** *x*∈# *relations-mset l* **using** *set-implies-mset* **by** *fast*
**then have** *0*: *count* (*relations-mset l*) *x* = *1* **using** *assms*(*1*) *distinct-rels-alt'* **by** *metis*
**have** *x*∉# *relations-mset r* **using** *True assms*(*3*) *disjoint-iff mset-implies-set* **by** *fast*
**then have** *count* (*relations-mset r*) *x* = *0* **by** (*simp add*: *count-eq-zero-iff*)
**then show** *?thesis* **using** *0* **by** *simp*
**next**
**case** *False*
**have** *x*∈# *relations-mset r* **using** *False assms*(*4*) **using** *mset-implies-set* **by** *force*
**then have** *0*: *count* (*relations-mset r*) *x* = *1* **using** *assms*(*2*) *distinct-rels-alt'* **by** *metis*

**have** *x∉# relations-mset l* **using** *False assms(3) disjoint-iff mset-implies-set*
**by** *fast*
**then have** *count* (*relations-mset l*) *x* = *0* **by** (*simp add*: *count-eq-zero-iff*)
**then show** *?thesis* **using** *0* **by** *simp*
**qed**
**qed**

**lemma** *distinct-and-disjoint-impl-distinct*:
⟦*distinct-relations l*; *distinct-relations r*; *relations l* ∩ *relations r* = {}⟧
⟹ *distinct-relations* (*Join l r*)
**using** *distinct-and-disjoint-impl-count1 distinct-rels-alt'* **by** *fastforce*

**lemma** *reasonable-trans*:
*reasonable-cards cf f* (*Join l r*) ⟹ *reasonable-cards cf f l* ∧ *reasonable-cards cf f r*
**by** (*simp add*: *Let-def*)

**lemma** *mirror-height-eq*: *height* (*mirror t*) = *height t*
**by**(*induction t*) *auto*

**lemma** *height-0-rel*: *height t* = *0* ⟹ ∃ *r*. *t* = *Relation r*
**by**(*cases t*) *auto*

**lemma** *height-gt-0-join*: *height t* > *0* ⟹ ∃ *l r*. *t* = *Join l r*
**by**(*cases t*) *auto*

**lemma** *height-decr-l*: *height* (*Join l r*) > *height l*
**by** *simp*

**lemma** *height-decr-r*: *height* (*Join l r*) > *height r*
**by** *simp*

**lemma** *mirror-num-relations-eq*: *num-relations* (*mirror t*) = *num-relations t*
**by**(*induction t*) *auto*

**lemma** *zig-zag-num-relations-height*: *zig-zag t* ⟹ *num-relations t* = *height t* + *1*
**by**(*induction t rule*: *zig-zag.induct*) *auto*

**lemma** *ldeep-num-relations-height*: *left-deep t* ⟹ *num-relations t* = *height t* + *1*
**by** (*simp add*: *zig-zag-num-relations-height ldeep-zig-zag*)

**lemma** *rdeep-num-relations-height*: *right-deep t* ⟹ *num-relations t* = *height t* + *1*
**by** (*simp add*: *zig-zag-num-relations-height rdeep-zig-zag*)

**lemma** *num-relations-eq-length*: *num-relations t* = *length* (*inorder t*)
**by**(*induction t*) *auto*

**lemma** *reasonable-impl-pos*: *reasonable-cards cf f t* ⟹ *pos-rel-cards cf t*

**by**(*induction t*) (*auto simp*: *pos-rel-cards-def Let-def*)

**lemma** *cards-list-eq-inorder*: *map* (λ(*a*,-). *a*) (*cards-list cf t*) = *inorder t*
**by**(*induction t*) *auto*

**lemma** *cards-list-eq-relations*: (λ(*a*,-). *a*) ' *set* (*cards-list cf t*) = *relations t*
**by** (*simp add*: *cards-list-eq-inorder image-set inorder-eq-set*)

**lemma** *cards-eq-c*: (*rel*,*c*)∈*set*(*cards-list cf t*) ⟹ *cf rel* = *c*
**by**(*induction t*) *auto*

**lemma** *finite-trans*: *finite* (*relations* (*Join l r*)) ⟹ *finite* (*relations l*) ∧ *finite* (*relations r*)
**by** *simp*

**lemma** *distinct-impl-card-eq-length*:
*finite* (*relations t*) ⟹ *height t* ≤ *n* ⟹ *distinct-relations t*
⟹ *Finite-Set.card* (*relations t*) = *length* (*inorder t*)
**proof**(*induction n arbitrary*: *t*)
**case** *0*
**then obtain** *r* **where** *Relation r* = *t* **using** *height-0-rel* **by** *auto*
**then show** *?case* **using** *distinct-relations-def* **by** *force*
**next**
**case** (*Suc n*)
**then show** *?case*
**proof**(*cases height t* = *Suc n*)
**case** *True*
**then have** *0* < *height t* **by** *simp*
**then obtain** *l r* **where** *join*[*simp*]: *Join l r* = *t* **using** *height-gt-0-join* **by** *blast*
**then have** *0*: *finite* (*relations l*) ∧ *finite* (*relations r*)
**using** *Suc.prems*(*1*) *finite-trans* **by** *blast*
**have** *1*: *height l* ≤ *n* **using** *True join* **by** (*metis height-decr-l less-Suc-eq-le*)
**have** *2*: *height r* ≤ *n* **using** *True join* **by** (*metis height-decr-r less-Suc-eq-le*)
**have** *Finite-Set.card* (*relations t*) + *Finite-Set.card* (*relations l* ∩ *relations r*)
= *Finite-Set.card* (*relations l*) + *Finite-Set.card* (*relations r*)
**using** *card-Un-Int join 0* **by** (*metis JoinTree.joinTree.simps*(*16*))
**then have** *Finite-Set.card* (*relations t*)
= *Finite-Set.card* (*relations l*) + *Finite-Set.card* (*relations r*)
**by** (*simp add*: *local.Suc.prems*(*3*) *distinct-relations-disjoint*)
**moreover have** *length* (*inorder t*)
= *length* (*inorder l*) + *length* (*inorder r*)
**by** (*metis JoinTree.inorder.simps*(*2*) *join length-append*)
**moreover have** *Finite-Set.card* (*relations l*) = *length* (*inorder l*)
**using** *Suc.IH Suc.prems*(*3*) *distinct-trans-l 0 1 join* **by** *blast*
**moreover have** *Finite-Set.card* (*relations r*) = *length* (*inorder r*)
**using** *Suc.IH Suc.prems*(*3*) *distinct-trans-r 0 2 join* **by** *blast*
**ultimately show** *?thesis* **by** *simp*
**next**
**case** *False*

**then show** *?thesis* **using** *Suc* **by** *simp*
**qed**
**qed**

**lemma** *card-le-length*: *Finite-Set.card* (*relations t*) ≤ *length* (*inorder t*)
**apply**(*induction t*)
**apply**(*auto*)[*2*]
**by** (*meson add-mono card-Un-le le-trans*)

**lemma** *card-eq-length-impl-disjunct*:
**assumes** *finite* (*relations* (*Join l r*))
**and** *Finite-Set.card* (*relations* (*Join l r*)) = *length* (*inorder* (*Join l r*))
**shows** *relations l* ∩ *relations r* = {}
**proof** (*rule ccontr*)
**assume** *0*: *relations l* ∩ *relations r* ≠ {}
**have** *1*: *finite* (*relations l*) ∧ *finite* (*relations r*) **using** *assms*(*1*) **by** *simp*
**then have** *2*: *Finite-Set.card* (*relations* (*Join l r*)) + *Finite-Set.card* (*relations l* ∩ *relations r*)
= *Finite-Set.card* (*relations l*) + *Finite-Set.card* (*relations r*)
**using** *card-Un-Int* **by** (*metis JoinTree.joinTree.simps*(*16*))
**moreover have** *Finite-Set.card* (*relations l* ∩ *relations r*) > *0* **using** *0 1* **by** *auto*
**ultimately have** *Finite-Set.card* (*relations* (*Join l r*))
< *Finite-Set.card* (*relations l*) + *Finite-Set.card* (*relations r*) **by** *simp*
**also have** ... ≤ *length* (*inorder l*) + *Finite-Set.card* (*relations r*)
**by** (*simp add*: *card-le-length*)
**also have** ... ≤ *length* (*inorder l*) + *length* (*inorder r*)
**by** (*simp add*: *card-le-length*)
**finally have** *Finite-Set.card* (*relations* (*Join l r*)) < *length* (*inorder* (*Join l r*))
**by** *simp*
**then show** *False* **using** *assms*(*2*) **by** *simp*
**qed**

**lemma** *card-eq-length-trans-l*:
**assumes** *finite* (*relations* (*Join l r*))
**and** *Finite-Set.card* (*relations* (*Join l r*)) = *length* (*inorder* (*Join l r*))
**shows** *Finite-Set.card* (*relations l*) = *length* (*inorder l*)
**proof** (*rule ccontr*)
**assume** *0*: *Finite-Set.card* (*relations l*) ≠ *length* (*inorder l*)
**have** *Finite-Set.card* (*relations* (*Join l r*))
= *length* (*inorder l*) + *length* (*inorder r*)
**using** *assms*(*2*) **by** *simp*
**have** *finite* (*relations l*) ∧ *finite* (*relations r*) **using** *assms*(*1*) **by** *simp*
**then have** *Finite-Set.card* (*relations* (*Join l r*)) + *Finite-Set.card* (*relations l* ∩ *relations r*)
= *Finite-Set.card* (*relations l*) + *Finite-Set.card* (*relations r*)
**using** *card-Un-Int* **by** (*metis JoinTree.joinTree.simps*(*16*))
**then have** *Finite-Set.card* (*relations* (*Join l r*))
= *Finite-Set.card* (*relations l*) + *Finite-Set.card* (*relations r*)

```
    using assms by (simp add: card-eq-length-impl-disjunct)
  moreover have Finite-Set.card (relations l) < length (inorder l)
    using 0 card-le-length le-imp-less-or-eq by blast
  ultimately have Finite-Set.card (relations (Join l r))
          < length (inorder l) + Finite-Set.card (relations r)
    by simp
  also have ... ≤ length (inorder l) + length (inorder r)
    by (simp add: card-le-length)
  finally have Finite-Set.card (relations (Join l r)) < length (inorder (Join l r))
    by simp
  then show False using assms(2) by simp
qed

lemma card-eq-length-trans-r:
  assumes finite (relations (Join l r))
      and Finite-Set.card (relations (Join l r)) = length (inorder (Join l r))
    shows Finite-Set.card (relations r) = length (inorder r)
  using assms card-eq-length-trans-l mirror-set-eq
 by (metis JoinTree.mirror.simps(2) mirror-num-relations-eq num-relations-eq-length)

lemma card-eq-length-impl-distinct:
  ⟦finite (relations t); height t ≤ n; Finite-Set.card (relations t) = length (inorder
t)⟧
    ⟹ distinct-relations t
proof(induction n arbitrary: t)
  case 0
  then obtain r where Relation r = t using height-0-rel by auto
  then show ?case using distinct-relations-def by force
next
  case (Suc n)
  then show ?case
  proof(cases height t = Suc n)
    case True
    then have 0 < height t by simp
   then obtain l r where join[simp]: Join l r = t using height-gt-0-join by blast
    then have 0: finite (relations l) ∧ finite (relations r)
      using Suc.prems(1) finite-trans by blast
    have 1: height l ≤ n using True join by (metis height-decr-l less-Suc-eq-le)
    have 2: height r ≤ n using True join by (metis height-decr-r less-Suc-eq-le)
    have Finite-Set.card (relations t) + Finite-Set.card (relations l ∩ relations r)
            = Finite-Set.card (relations l) + Finite-Set.card (relations r)
      using card-Un-Int join 0 by (metis JoinTree.joinTree.simps(16))
    then have Finite-Set.card (relations t)
            = Finite-Set.card (relations l) + Finite-Set.card (relations r)
      using Suc.prems(1,3) by (simp add: card-eq-length-impl-disjunct)

    have Finite-Set.card (relations l) = length (inorder l)
      using Suc.prems(1,3) card-eq-length-trans-l join by blast
    then have 3: distinct-relations l using Suc.IH 0 1 by blast
```

```
    have Finite-Set.card (relations r) = length (inorder r)
      using Suc.IH Suc.prems(1,3) card-eq-length-trans-r join by blast
    then have 4: distinct-relations r using Suc.IH 0 2 by blast
    have relations l ∩ relations r = {}
      using card-eq-length-impl-disjunct join Suc.prems(1,3) by blast
    then show ?thesis using 3 4 distinct-and-disjoint-impl-distinct by fastforce
  next
    case False
    then show ?thesis using Suc by simp
  qed
qed

lemma list-sel-revorder-eq-inorder-x: list-sel f (revorder l) ys = list-sel f (inorder
l) ys
  unfolding revorder-eq-rev-inorder using mset-x-eq-list-sel-eq mset-rev by blast

lemma list-sel-revorder-eq-inorder-y: list-sel f xs (revorder r) = list-sel f xs (inorder
r)
  unfolding revorder-eq-rev-inorder using mset-y-eq-list-sel-eq mset-rev by blast

lemma list-sel-revorder-eq-inorder:
  list-sel f (revorder l) (revorder r) = list-sel f (inorder l) (inorder r)
  unfolding list-sel-revorder-eq-inorder-x list-sel-revorder-eq-inorder-y by simp

lemma card-join-alt:
  card cf f (Join l r) = list-sel f (revorder l) (revorder r) * card cf f l * card cf f r
  unfolding list-sel-revorder-eq-inorder by simp

lemma distinct-alt:
  finite (relations t)
    ⟹ distinct-relations t ⟷ Finite-Set.card (relations t) = length (inorder t)
  using card-eq-length-impl-distinct distinct-impl-card-eq-length by auto

lemma distinct-alt2:
  distinct-relations (Join l r)
    ⟷ distinct-relations l ∧ distinct-relations r ∧ relations l ∩ relations r = {}
  using distinct-relations-disjoint distinct-trans-l distinct-trans-r
  by (auto elim: distinct-and-disjoint-impl-distinct)

lemma pos-rel-cards-subtrees:
  pos-rel-cards cf (Join l r) = (pos-rel-cards cf l ∧ pos-rel-cards cf r)
proof −
  have pos-rel-cards cf (Join l r) = (∀(-,c)∈set (cards-list cf (Join l r)). c>0)
    by (simp add: pos-rel-cards-def)
  also have ... = (∀(-,c)∈set (cards-list cf l @ cards-list cf r). c>0) by simp
  also have ... = ((∀(-,c)∈set (cards-list cf l). c>0) ∧ (∀(-,c)∈set (cards-list cf
r). c>0))
    by auto
  also have ... = (pos-rel-cards cf l ∧ pos-rel-cards cf r)
```

```
    by (simp add: pos-rel-cards-def)
  finally show ?thesis by simp
qed

lemma pos-rel-cards-eq-pos-list-cards:
  pos-rel-cards cf t ⟷ pos-list-cards cf (inorder t)
  by(induction t) (auto simp: pos-rel-cards-def pos-list-cards-def)

lemma pos-list-cards-split:
  pos-list-cards cf (xs@ys) ⟷ pos-list-cards cf xs ∧ pos-list-cards cf ys
  by(induction xs) (auto simp: pos-list-cards-def)

lemma pos-sel-reason-impl-reason:
  ⟦pos-rel-cards cf t; sel-reasonable sel⟧ ⟹ reasonable-cards cf sel t
proof(induction t)
  case (Join l r)
  then have pos-rel-cards cf l ∧ pos-rel-cards cf r using pos-rel-cards-subtrees by
blast
  then have 0: reasonable-cards cf sel l ∧ reasonable-cards cf sel r using Join by
simp
  have list-sel sel (inorder l) (inorder r) ≤ 1
    using Join.prems(2) sel-reasonable-def list-sel-reasonable by fast
  obtain c where 1:
    list-sel sel (inorder l) (inorder r) * card cf sel l * card cf sel r = c
    by simp
  then have c = list-sel sel (inorder l) (inorder r) * card cf sel l * card cf sel r
    by simp
  then have 2: c ≤ 1 * card cf sel l * card cf sel r
   using Join.prems(2) list-sel-reasonable 0 mult-left-le-one-le mult-right-less-imp-less
   by (smt (verit, ccfv-SIG) card.simps(1) card.simps(2) reasonable-cards.elims(2))
  from 1 have c > 0 * card cf sel l * card cf sel r
    using Join.prems(2) list-sel-reasonable 0 mult-pos-pos
    by (metis card.simps(1) card.simps(2) mult-eq-0-iff reasonable-cards.elims(2))
  then show ?case using 0 1 2 by simp
qed(simp add: pos-rel-cards-def)

lemma create-rdeep-order: xs ≠ [] ⟹ inorder (create-rdeep xs) = xs
proof(induction xs)
  case (Cons x xs)
  then show ?case by(cases xs) auto
qed(simp)

lemma create-ldeep-rev-order: xs ≠ [] ⟹ inorder (create-ldeep-rev xs) = rev xs
proof(induction xs)
  case (Cons x xs)
  then show ?case by(cases xs) auto
qed(simp)

lemma create-ldeep-order: xs ≠ [] ⟹ inorder (create-ldeep xs) = xs
```

**by** (*simp add*: *create-ldeep-def create-ldeep-rev-order*)

**lemma** *create-rdeep-rdeep*: *xs* ≠ [] ⟹ *right-deep* (*create-rdeep xs*)
**proof**(*induction xs*)
**case** (*Cons x xs*)
**then show** *?case* **by**(*cases xs*) *auto*
**qed**(*simp*)

**lemma** *create-ldeep-rev-ldeep*: *xs* ≠ [] ⟹ *left-deep* (*create-ldeep-rev xs*)
**proof**(*induction xs*)
**case** (*Cons x xs*)
**then show** *?case* **by**(*cases xs*) *auto*
**qed**(*simp*)

**lemma** *create-ldeep-ldeep*: *xs* ≠ [] ⟹ *left-deep* (*create-ldeep xs*)
**by** (*simp add*: *create-ldeep-rev-ldeep create-ldeep-def*)

**lemma** *create-ldeep-rev-relations*: *xs* ≠ [] ⟹ *relations* (*create-ldeep-rev xs*) = *set xs*
**using** *create-ldeep-rev-order*[*of xs*] *inorder-eq-set* **by** *force*

**lemma** *create-ldeep-relations*: *xs* ≠ [] ⟹ *relations* (*create-ldeep xs*) = *set xs*
**by** (*simp add*: *create-ldeep-rev-relations create-ldeep-def*)

**lemma** *create-ldeep-rev-Cons*:
*xs* ≠ [] ⟹ *create-ldeep-rev* (*x*#*xs*) = *Join* (*create-ldeep-rev xs*) (*Relation x*)
**using** *create-ldeep-rev.simps*(*3*) *neq-Nil-conv* **by** *metis*

**lemma** *create-ldeep-snoc*: *xs* ≠ [] ⟹ *create-ldeep* (*xs*@[*x*]) = *Join* (*create-ldeep xs*) (*Relation x*)
**by** (*simp add*: *create-ldeep-rev-Cons create-ldeep-def*)

**lemma** *create-ldeep-inorder*[*simp*]: *left-deep t* ⟹ *create-ldeep* (*inorder t*) = *t*
**apply**(*induction t*)
**apply** (*simp add*: *create-ldeep-def*)
**by** (*metis Nil-is-append-conv create-ldeep-snoc inorder.simps*
*ldeep-trans left-deep.simps*(*3*) *not-Cons-self2 relations-mset.cases*)

**lemma** *create-rdeep-inorder*[*simp*]: *right-deep t* ⟹ *create-rdeep* (*inorder t*) = *t*
**apply**(*induction t*)
**apply** *simp*
**by** (*metis create-rdeep.simps*(*3*) *create-rdeep-order first-node-first-inorder*
*joinTree.distinct*(*1*) *joinTree.inject*(*2*) *neq-Nil-conv right-deep.elims*(*2*))

**lemma** *ldeep-div-eq-sel*:
**assumes** *reasonable-cards cf f* (*Join l* (*Relation rel*))
**and** *c* = *card cf f* (*Join l* (*Relation rel*))
**and** *cr* = *card cf f* (*Relation rel*)
**shows** *c* / (*card cf f l* ∗ *cr*) = *list-sel f* (*inorder l*) [*rel*]

**using** *assms* **by** *auto*

**lemma** *ldeep-n-eq-card*:
⟦*distinct-relations t*; *left-deep t*⟧ ⟹ *ldeep-n f cf* (*revorder t*) = *card cf f t*
**proof**(*induction t arbitrary*: *cf rule*: *left-deep.induct*)
**case** (*2 l rr*)
**let** *?rev* = *revorder* (*Join l* (*Relation rr*))
**have** *?rev* = *rr* # *revorder l* **by** *simp*
**have** *ldeep-n f cf ?rev* = *ldeep-n f cf* (*rr*#*revorder l*) **by** *simp*
**also have** ... = *list-sel-aux′ f* (*revorder l*) *rr*
∗ *cf rr* ∗ *ldeep-n f cf* (*revorder l*) **by** *simp*
**also have** ... = *list-sel-aux′ f* (*inorder l*) *rr* ∗ *cf rr*
∗ *ldeep-n f cf* (*revorder l*)
**using** *mset-x-eq-list-sel-aux′-eq mset-rev* **by** (*fastforce simp*: *revorder-eq-rev-inorder*)
**also have** ... = *list-sel-aux′ f* (*inorder l*) *rr* ∗ *cf rr* ∗ *card cf f l*
**using** *2 distinct-trans-l* **by** *auto*
**finally show** *?case*
**using** *list-sel-sing-aux′ card.simps mult.commute*
**by** (*metis ab-semigroup-mult-class.mult-ac*(*1*) *inorder.simps*(*1*))
**qed**(*auto*)

**lemma** *ldeep-n-eq-card-subtree*:
⟦*distinct-relations* (*Join t r′*); *left-deep t*⟧ ⟹ *ldeep-n f cf* (*revorder t*) = *card cf f t*
**using** *ldeep-n-eq-card distinct-trans-l* **by** *blast*

**lemma** *distinct-ldeep-T′-prepend*:
*distinct* (*ys*@*xs*) ⟹ *ldeep-T′* (*ldeep-s f* (*ys*@*xs*)) *cf xs* = *ldeep-T′* (*ldeep-s f xs*) *cf xs*
**proof**(*induction xs arbitrary*: *ys*)
**case** (*Cons x xs*)
**then have** *0*: *distinct* (*x*#*xs*) **by** *simp*
**have** *ldeep-T′* (*ldeep-s f* (*ys*@*x*#*xs*)) *cf* (*x*#*xs*)
= *cf x* ∗ (*ldeep-s f* (*ys*@*x*#*xs*)) *x* ∗ *ldeep-T′* (*ldeep-s f* (*ys*@*x*#*xs*)) *cf xs*
**by** *simp*
**also have** ... = *cf x* ∗ (*ldeep-s f* (*ys*@*x*#*xs*)) *x* ∗ *ldeep-T′* (*ldeep-s f xs*) *cf xs*
**using** *Cons.IH*[*of ys*@[*x*]] *Cons.prems* **by** *simp*
**also have** ... = *cf x* ∗ *list-sel-aux′ f xs x* ∗ *ldeep-T′* (*ldeep-s f xs*) *cf xs*
**using** *distinct-ldeep-s-eq-aux*[*OF Cons.prems*] **by** *simp*
**also have** ... = *cf x* ∗ (*ldeep-s f* (*x*#*xs*)) *x* ∗ *ldeep-T′* (*ldeep-s f xs*) *cf xs*
**using** *distinct-ldeep-s-eq-aux Cons.prems* **by** *simp*
**also have** ... = *cf x* ∗ (*ldeep-s f* (*x*#*xs*)) *x* ∗ *ldeep-T′* (*ldeep-s f* (*x*#*xs*)) *cf xs*
**using** *Cons.IH*[*of* [*x*]] *0* **by** *simp*
**finally show** *?case* **by** *simp*
**qed**(*simp*)

**lemma** *ldeep-T′-eq-ldeep-n*: *distinct xs* ⟹ *ldeep-T′* (*ldeep-s f xs*) *cf xs* = *ldeep-n f cf xs*

**proof**(*induction xs*)
  **case** (*Cons x xs*)
  **then have** *0*: *distinct xs* **by** *simp*
  **have** *ldeep-T′* (*ldeep-s f* (*x # xs*)) *cf* (*x # xs*)
        = *cf x ∗* (*ldeep-s f* (*x # xs*)) *x ∗ ldeep-T′* (*ldeep-s f* (*x # xs*)) *cf xs* **by** *simp*
  **also have** ... = *cf x ∗ list-sel-aux′ f xs x ∗ ldeep-T′* (*ldeep-s f* (*x # xs*)) *cf xs* **by** *simp*
  **also have** ... = *cf x ∗ list-sel-aux′ f xs x ∗ ldeep-T′* (*ldeep-s f xs*) *cf xs*
    **using** *distinct-ldeep-T′-prepend*[*of* [*x*]] *Cons.prems* **by** *simp*
  **also have** ... = *cf x ∗ list-sel-aux′ f xs x ∗ ldeep-n f cf xs*
    **using** *Cons.IH 0* **by** *simp*
  **finally show** *?case* **by** *simp*
**qed**(*simp*)

**lemma** *ldeep-T′-eq-foldl*: *acc ∗ ldeep-T′ f cf xs* = *foldl* (*λa b. a ∗ cf b ∗ f b*) *acc xs*
**proof**(*induction xs arbitrary*: *acc*)
  **case** (*Cons x xs*)
  **have** *acc ∗ ldeep-T′ f cf* (*x # xs*) = *acc ∗ cf x ∗ f x ∗ ldeep-T′ f cf xs* **by** *simp*
  **also have** ... = *foldl* (*λa b. a ∗ cf b ∗ f b*) (*acc ∗ cf x ∗ f x*) *xs* **using** *Cons* **by** *simp*
  **finally show** *?case* **by** *simp*
**qed**(*simp*)

**lemma** *distinct-ldeep-T-prepend*:
  *distinct* (*ys*@*xs*) ⟹ *ldeep-T* (*ldeep-s f* (*ys*@*xs*)) *cf xs* = *ldeep-T* (*ldeep-s f xs*) *cf xs*
  **using** *ldeep-T′-eq-foldl*[*of 1 ldeep-s f* (*ys*@*xs*) *cf xs*]
  **by** (*simp add*: *distinct-ldeep-T′-prepend ldeep-T-def ldeep-T′-eq-foldl*)

**lemma** *ldeep-T-eq-ldeep-T′-aux*: *ldeep-T sf cf xs* = *ldeep-T′ sf cf xs*
  **using** *ldeep-T′-eq-foldl*[*of 1 sf*] *ldeep-T-def* **by** *fastforce*

**lemma** *ldeep-T-eq-ldeep-T′*: *ldeep-T* = *ldeep-T′*
  **using** *ldeep-T-eq-ldeep-T′-aux* **by** *blast*

**lemma** *ldeep-T-eq-ldeep-n*: *distinct xs* ⟹ *ldeep-T* (*ldeep-s f xs*) *cf xs* = *ldeep-n f cf xs*
  **by** (*simp add*: *ldeep-T-eq-ldeep-T′ ldeep-T′-eq-ldeep-n*)

**lemma** *ldeep-T-app*: *ldeep-T f cf* (*xs*@*ys*) = *ldeep-T f cf xs ∗ ldeep-T f cf ys*
  **using** *ldeep-T-def foldl-append ldeep-T′-eq-foldl*
  **by** (*metis* (*mono-tags*, *lifting*) *monoid.left-neutral mult.monoid-axioms*)

**lemma** *ldeep-T-empty*: *ldeep-T f cf* [] = *1*
  **by** (*simp add*: *ldeep-T-def*)

**lemma** *ldeep-T-eq-if-cf-eq*: ∀ *x* ∈ *set xs. f x* = *g x* ⟹ *ldeep-T sf f xs* = *ldeep-T sf g xs*

**unfolding** *ldeep-T-eq-ldeep-T'* **by** (*induction xs*) *auto*

**lemma** *ldeep-n-pos*: ⟦*pos-list-cards cf xs*; *sel-reasonable f*⟧ ⟹ *ldeep-n f cf xs* > *0*
**proof**(*induction xs*)
**case** *Nil*
**then show** *?case* **by** *simp*
**next**
**case** (*Cons x xs*)
**then show** *?case*
**using** *list-sel-aux'-reasonable pos-list-cards-def mult-pos-pos set-subset-Cons*
**by** (*metis list.set-intros*(*1*) *ldeep-n.simps*(*2*) *subset-code*(*1*))
**qed**

**lemma** *ldeep-T-eq-card*:
⟦*distinct-relations t*; *left-deep t*⟧
⟹ *ldeep-T* (*ldeep-s f* (*revorder t*)) *cf* (*revorder t*) = *card cf f t*
**using** *ldeep-T-eq-ldeep-n*[*of revorder t*] *ldeep-n-eq-card distinct-rels-alt* **by** *fastforce*

**lemma** *ldeep-T-pos'*:
⟦*distinct xs*; *pos-list-cards cf xs*; *sel-reasonable f*⟧ ⟹ *ldeep-T* (*ldeep-s f xs*) *cf xs* > *0*
**by** (*simp add*: *ldeep-T-eq-ldeep-n ldeep-n-pos*)

**lemma** *ldeep-T-pos*: ⟦∀ *x*∈ *set ys*. *cf x* > *0*; *sel-reasonable f*⟧ ⟹ *ldeep-T* (*ldeep-s f xs*) *cf ys* > *0*
**apply**(*induction ys arbitrary*: *xs*)
**apply**(*auto simp*: *ldeep-T-def*)[*2*]
**by** (*metis Groups.comm-monoid-mult-class.mult-1 ldeep-T'-eq-foldl ldeep-s-pos zero-less-mult-iff*)

**end**

**theory** *CostFunctions*
**imports** *Complex-Main JoinTree Selectivities*
**begin**

# 3 Cost Functions

## 3.1 General Cost Functions

**fun** *c-out* :: *'a card* ⇒ *'a selectivity* ⇒ *'a joinTree* ⇒ *real* **where**
*c-out - -* (*Relation -*) = *0*
| *c-out cf f* (*Join l r*) = *card cf f* (*Join l r*) + *c-out cf f l* + *c-out cf f r*

**fun** *c-nlj* :: *'a card* ⇒ *'a selectivity* ⇒ *'a joinTree* ⇒ *real* **where**
*c-nlj - -* (*Relation -*) = *0*
| *c-nlj cf f* (*Join l r*) = *card cf f l* ∗ *card cf f r* + *c-nlj cf f l* + *c-nlj cf f r*

**fun** *c-hj* :: *'a card* ⇒ *'a selectivity* ⇒ *'a joinTree* ⇒ *real* **where**
  *c-hj - - (Relation -) = 0*
| *c-hj cf f (Join l r) = 1.2 * card cf f l + c-hj cf f l + c-hj cf f r*

**fun** *c-smj* :: *'a card* ⇒ *'a selectivity* ⇒ *'a joinTree* ⇒ *real* **where**
  *c-smj - - (Relation -) = 0*
| *c-smj cf f (Join l r) = card cf f l * log 2 (card cf f l) + card cf f r * log 2 (card cf f r)*
    *+ c-smj cf f l + c-smj cf f r*

## 3.2 Cost functions that are considered by IKKBZ.

**fun** *c-IKKBZ* :: *('a ⇒ real ⇒ real) ⇒ 'a card ⇒ 'a selectivity ⇒ 'a joinTree ⇒ real* **where**
  *c-IKKBZ - - - (Relation -) = 0*
| *c-IKKBZ h cf f (Join l (Relation rel)) = card cf f l * (h rel (cf rel)) + c-IKKBZ h cf f l*
| *c-IKKBZ - - - (Join l r) = undefined*

A list of relations defines a unique left-deep tree. This functions computes a cost function given by such a list representation of a tree according to the formula $\sum_{i=2}^{n} n_{\{1,2,\ldots i-1\}}\ h_i(n_i)$ where $n_{\{1,2,\ldots i-1\}} = JoinTree.card\ subtree = ldeep\text{-}n\ f\ cf$ (list subtree) The input list is expected to be in reversed order for easier recursive processing i.e. the first element in xs is the rightmost element of the left-deep tree

**fun** *c-list'* :: *'a selectivity* ⇒ *'a card* ⇒ *('a list ⇒ 'a ⇒ real) ⇒ 'a list ⇒ real* **where**
  *c-list' - - - [] = 0*
| *c-list' - - - [x] = 0*
| *c-list' f cf h (x#xs) = ldeep-n f cf xs * h xs x + c-list' f cf h xs*

Equivalent definition which allows splitting the list at any point.

**fun** *c-list* :: *('a ⇒ real) ⇒ 'a card ⇒ ('a ⇒ real) ⇒ 'a ⇒ 'a list ⇒ real* **where**
  *c-list - - - - [] = 0*
| *c-list - - h r [x] = (if x=r then 0 else h x)*
| *c-list sf cf h r (x#xs) = c-list sf cf h r xs + ldeep-T sf cf xs * c-list sf cf h r [x]*

Maps the h function to a static version that doesn't require an input list.

**fun** *create-h-list* :: *('a list ⇒ 'a ⇒ real) ⇒ 'a list ⇒ 'a ⇒ real* **where**
  *create-h-list - [] = (λ-. 1)*
| *create-h-list h (x#xs) = (λa. if a=x then h xs x else create-h-list h xs a)*

## 3.3 Properties of Cost Functions

**definition** *symmetric* :: *('a joinTree ⇒ real) ⇒ bool* **where**
  *symmetric f = (∀ x y. f (Join x y) = f (Join y x))*

**definition** *symmetric'* :: (*'a card* ⇒ *'a selectivity* ⇒ *'a joinTree* ⇒ *real*) ⇒ *bool* **where**
*symmetric' f* = (∀ *x y cf sf*. *sel-symm sf* ⟶ (*f cf sf* (*Join x y*) = *f cf sf* (*Join y x*)))

Uses reversed lists since the last joined relation should only appear once. Therefore, it should be the head of the list and by inductive reasoning the list should be reversed. Furthermore, the root must be the first relation in the sequence (last in the reverse) or it must not be contained at all.

**definition** *asi'* :: *'a* ⇒ (*'a list* ⇒ *real*) ⇒ *bool* **where**
*asi' r c* = (∃ *rank* :: (*'a list* ⇒ *real*).
(∀ *A U V B*. *distinct* (*A*@*U*@*V*@*B*) ∧ *U* ≠ [] ∧ *V* ≠ []
∧ (*r* ∉ *set* (*A*@*U*@*V*@*B*) ∨ (*take 1* (*A*@*U*@*V*@*B*) = [*r*] ∧ *take 1* (*A*@*V*@*U*@*B*) = [*r*]))
⟶ (*c* (*rev* (*A*@*U*@*V*@*B*)) ≤ *c* (*rev* (*A*@*V*@*U*@*B*)) ⟷ *rank* (*rev U*) ≤ *rank* (*rev V*))))

**definition** *asi* :: (*'a list* ⇒ *real*) ⇒ *'a* ⇒ (*'a list* ⇒ *real*) ⇒ *bool* **where**
*asi rank r c* = (∀ *A U V B*. *distinct* (*A*@*U*@*V*@*B*) ∧ *U* ≠ [] ∧ *V* ≠ []
∧ (*r* ∉ *set* (*A*@*U*@*V*@*B*) ∨ (*take 1* (*A*@*U*@*V*@*B*) = [*r*] ∧ *take 1* (*A*@*V*@*U*@*B*) = [*r*]))
⟶ (*c* (*rev* (*A*@*U*@*V*@*B*)) ≤ *c* (*rev* (*A*@*V*@*U*@*B*)) ⟷ *rank* (*rev U*) ≤ *rank* (*rev V*)))

**definition** *asi''* :: (*'a list* ⇒ *real*) ⇒ *'a* ⇒ (*'a list* ⇒ *real*) ⇒ *bool* **where**
*asi'' rank r c* = ((∀ *A U V B*. *distinct* (*A*@*U*@*V*@*B*) ∧ *U* ≠ [] ∧ *V* ≠ [] ∧ *U* ≠ [*r*] ∧ *V* ≠ [*r*]
⟶ (*c* (*rev* (*A*@*U*@*V*@*B*)) ≤ *c* (*rev* (*A*@*V*@*U*@*B*)) ⟷ *rank* (*rev U*) ≤ *rank* (*rev V*))))

## 3.4 Proofs

**lemma** *c-out-symm*: *sel-symm f* ⟹ *symmetric* (*c-out cf f*)
**by** (*simp add*: *symmetric-def list-sel-symm*)

**lemma** *c-nlj-symm*: *symmetric* (*c-nlj cf f*)
**by** (*simp add*: *symmetric-def*)

**lemma** *c-smj-symm*: *symmetric* (*c-smj cf f*)
**by** (*simp add*: *symmetric-def*)

### 3.4.1 Equivalence Proofs

**theorem** *c-nlj-IKKBZ*: *left-deep t* ⟹ *c-nlj cf f t* = *c-IKKBZ* (λ-. *id*) *cf f t*
**proof**(*induction t*)
**case** (*Join l r*)
**then show** *?case* **by**(*cases r*) *auto*
**qed**(*simp*)

**theorem** *c-hj-IKKBZ*: *left-deep t* $\Longrightarrow$ *c-hj cf f t* = *c-IKKBZ* (*λ- -. 1.2*) *cf f t*
**proof**(*induction t*)
**case** *ind*: (*Join l r*)
**then show** *?case* **by**(*cases r*) *auto*
**qed**(*simp*)

**lemma** *change-fun-order*: *y≠rel*
$\Longrightarrow$ (*λa b. if a=rel then g a b else* (*λc d. if c=y then h c d else f c d*) *a b*)
= (*λa b. if a=y then h a b else* (*λc d. if c=rel then g c d else f c d*) *a b*)
**by** *fastforce*

**lemma** *c-IKKBZ-fun-notelem*:
**assumes** *left-deep t*
**and** *distinct-relations t*
**and** *y* $\notin$ *relations t*
**and** *f′* = (*λa b. if a=y then z b else f a b*)
**shows** *c-IKKBZ f′ cf sf t* = *c-IKKBZ f cf sf t*
**using** *assms* **proof**(*induction t arbitrary*: *f′ f z rule*: *left-deep.induct*)
**case** (*2 l rel*)
**then have** *0*: *rel* $\neq$ *y* **by** *auto*
**have** *c-IKKBZ f′ cf sf* (*Join l* (*Relation rel*))
= *card cf sf l* $*$ (*f′ rel* (*cf rel*)) + *c-IKKBZ f′ cf sf l* **by** *simp*
**also have** ... = *card cf sf l* $*$ (*f′ rel* (*cf rel*)) + *c-IKKBZ f cf sf l*
**using** *ldeep-trans distinct-trans-l 2* **by** *fastforce*
**also have** ... = *card cf sf l* $*$ (*f rel* (*cf rel*)) + *c-IKKBZ f cf sf l*
**using** *2.prems*(*3*,*4*) **by** *fastforce*
**also have** ... = *c-IKKBZ f cf sf* (*Join l* (*Relation rel*)) **using** *2.prems*(*1*) **by** *simp*
**finally show** *?case* **.**
**qed** (*auto*)

**lemma** *distinct-c-IKKBZ-ldeep-s-prepend*:
⟦*distinct*(*ys@revorder t*); *left-deep t*⟧
$\Longrightarrow$ *c-IKKBZ* (*λa b. ldeep-s f* (*ys@revorder t*) *a* $*$ *b*) *cf f t*
= *c-IKKBZ* (*λa b. ldeep-s f* (*revorder t*) *a* $*$ *b*) *cf f t*
**proof**(*induction t arbitrary*: *ys rule*: *left-deep.induct*)
**case** (*2 l rr*)
**let** *?ylr* = *ys* @ *revorder* (*Join l* (*Relation rr*))
**let** *?lr* = *revorder* (*Join l* (*Relation rr*))
**let** *?h* = (*λa.* (*∗*) (*ldeep-s f ?ylr a*))
**let** *?h′* = (*λa.* (*∗*) (*ldeep-s f ?lr a*))
**let** *?h″* = (*λa.* (*∗*) (*ldeep-s f* (*revorder l*) *a*))
**have** *?lr* = [*rr*]@*revorder l* **by** *simp*
**have** *0*: *distinct ?lr* **using** *2.prems*(*1*) **by** *simp*
**have** *c-IKKBZ ?h cf f* (*Join l* (*Relation rr*))
= *card cf f l* $*$ ((*ldeep-s f ?ylr rr*) $*$ (*cf rr*)) + *c-IKKBZ ?h cf f l*
**by** *simp*
**also have** ... = *card cf f l* $*$ ((*list-sel-aux′ f* (*revorder l*) *rr*) $*$ (*cf rr*))

```
          + c-IKKBZ ?h cf f l
    using 2.prems(1) by (fastforce simp: distinct-ldeep-s-eq-aux)
  also have ... = card cf f l * (?h' rr (cf rr)) + c-IKKBZ ?h cf f l by simp
  also have ... = card cf f l * (?h' rr (cf rr)) + c-IKKBZ ?h'' cf f l
    using 2.IH[of ys@[rr]] 2.prems by simp
  also have ... = card cf f l * (?h' rr (cf rr)) + c-IKKBZ ?h' cf f l
    using 2.IH[of [rr]] 2.prems(2) 0 by simp
  finally show ?case by simp
qed (auto)

lemma distinct-c-IKKBZ-ldeep-s-subtree:
  assumes distinct-relations (Join l (Relation rel))
      and left-deep (Join l (Relation rel))
    shows c-IKKBZ (λa b. ldeep-s f (revorder (Join l (Relation rel))) a * b) cf f l
        = c-IKKBZ (λa b. ldeep-s f (revorder l) a * b) cf f l
proof −
  have distinct (revorder (Join l (Relation rel)))
    using assms(1) by (simp add: distinct-rels-alt inorder-eq-mset)
  then have distinct ([rel]@revorder l) by simp
  then show ?thesis using distinct-c-IKKBZ-ldeep-s-prepend[of [rel] l] assms(2)
by simp
qed

theorem c-out-IKKBZ:
  ⟦distinct-relations t; reasonable-cards cf f t; left-deep t⟧
    ⟹ c-IKKBZ (λa b. ldeep-s f (revorder t) a * b) cf f t = c-out cf f t
proof(induction t)
  case ind: (Join l r)
  then show ?case
  proof(cases r)
    case (Relation rel)
    let ?s = (λa b. ldeep-s f (revorder (Join l r)) a * b)
    let ?s' = (λa b. ldeep-s f (revorder l) a * b)
    have c-IKKBZ ?s cf f l = c-IKKBZ ?s' cf f l
      using ind.prems distinct-c-IKKBZ-ldeep-s-subtree Relation by fast
    then have 0: c-IKKBZ ?s cf f l = c-out cf f l
      using ind ldeep-trans distinct-trans-l reasonable-trans by metis
    have c-IKKBZ ?s cf f (Join l r) = card cf f l * (?s rel (cf rel)) + c-IKKBZ ?s
cf f l
      using Relation by simp
    also have ... = card cf f l * ((list-sel-aux' f (revorder l) rel) * (cf rel))
            + c-IKKBZ ?s cf f l
      using Relation by simp
    also have ... = card cf f l * ((list-sel f (revorder l) [rel]) * (cf rel))
            + c-IKKBZ ?s cf f l
      by (simp add: list-sel-sing-aux')
    also have ... = card cf f l * ((list-sel f (inorder l) [rel]) * (cf rel))
            + c-IKKBZ ?s cf f l
     using mset-x-eq-list-sel-eq[of revorder l] by (simp add: revorder-eq-rev-inorder)
```

```
    also have ... = card cf f (Join l r) + c-IKKBZ ?s' cf f l
      using distinct-c-IKKBZ-ldeep-s-subtree ind.prems Relation by fastforce
    also have ... = card cf f (Join l r) + c-out cf f l
      using ind reasonable-trans distinct-trans-l ldeep-trans by metis
    finally show ?thesis using Relation by simp
  next
    case (Join lr rr)
    then show ?thesis using ind by simp
  qed
qed(simp)

theorem c-out-eq-c-list':
  ⟦distinct-relations t; reasonable-cards cf f t; left-deep t⟧
    ⟹ c-list' f cf (λxs x. (list-sel-aux' f xs x) ∗ cf x) (revorder t) = c-out cf f t
proof(induction t rule: left-deep.induct)
  case (2 l rr)
  let ?h = λxs x. list-sel-aux' f xs x ∗ cf x
  let ?ll = revorder l
  have 1: distinct-relations l using 2.prems distinct-trans-l by simp
  have 2: reasonable-cards cf f l using 2.prems reasonable-trans by blast
  have 3: left-deep l using 2.prems by simp
  have revorder (Join l (Relation rr)) = rr # ?ll by simp
  then have c-list' f cf ?h (revorder (Join l (Relation rr)))
        = ldeep-n f cf ?ll ∗ ?h ?ll rr + c-list' f cf ?h ?ll
    using joinTree-cases-ldeep[OF 3] by auto
  also have ... = card cf f l ∗ ?h ?ll rr + c-list' f cf ?h ?ll
    using ldeep-n-eq-card-subtree 2.prems by auto
  also have ... = card cf f l ∗ (list-sel-aux' f ?ll rr) ∗ cf rr + c-list' f cf ?h ?ll
    using mset-x-eq-list-sel-aux'-eq mset-rev by fastforce
  also have ... = card cf f (Join l (Relation rr)) + c-list' f cf ?h ?ll
    unfolding card-join-alt by (simp add: list-sel-sing-aux')
  also have ... = card cf f (Join l (Relation rr)) + c-out cf f l using 2.IH 1 2 3
by simp
  finally show ?case by simp
qed (auto)

lemma rev-first-last-elem: (rev (x#x'#xs')) = (r#rs) ⟹ x ∈# mset rs
  using  in-multiset-in-set last-in-set last-snoc rev-singleton-conv
  by (metis List.last.simps List.list.discI List.list.inject List.rev.simps(2))

lemma distinct-first-uneq-last: distinct (x#x'#xs') ⟹ rev (x#x'#xs') = r#rs
⟹ r ≠ x
  using rev-first-last-elem mset-rev set-mset-mset
  by (metis List.distinct.simps(2) count-eq-zero-iff distinct-count-atmost-1)

lemma distinct-create-eq-app:
  ⟦distinct (ys@xs); x ∈# mset xs⟧ ⟹ create-h-list h xs x = create-h-list h (ys@xs)
x
  by(induction ys) auto
```

**lemma** *c-list-single-h-list-not-elem-prepend*:
*x* ∉ *set ys*
⟹ *c-list f cf* (*create-h-list h* (*ys*@*x*#*xs*)) *r* [*x*] = *c-list f cf* (*create-h-list h* (*x*#*xs*)) *r* [*x*]
**by**(*induction ys*) *auto*

**lemma** *c-list-single-f-list-not-elem-prepend*:
*x* ∉ *set ys*
⟹ *c-list* (*ldeep-s f* (*ys*@*x*#*xs*)) *cf h r* [*x*] = *c-list* (*ldeep-s f* (*x*#*xs*)) *cf h r* [*x*]
**by**(*induction ys*) *auto*

**lemma** *c-list-prepend-h-disjunct*:
**assumes** *distinct* (*ys*@*xs*)
**shows** *c-list f cf* (*create-h-list h* (*ys*@*xs*)) *r xs* = *c-list f cf* (*create-h-list h xs*) *r xs*
**using** *assms* **proof**(*induction xs arbitrary*: *ys*)
**case** (*Cons x xs*)
**then have** *0*: *distinct* (*ys* @ [*x*] @ *xs*) **by** *simp*
**then have** *1*: *distinct* ([*x*] @ *xs*) **by** *simp*
**let** *?h* = *create-h-list h* (*ys* @ *x* # *xs*)
**let** *?h'* = *create-h-list h xs*
**let** *?h''* = *create-h-list h* (*x*#*xs*)
**have** *2*: *x* ∉ *set ys* **using** *Cons.prems* **by** *simp*
**show** *?case*
**proof**(*cases xs*=[])
**case** *True*
**then show** *?thesis*
**using** *Cons distinct-create-eq-app in-multiset-in-set*
**by** (*metis CostFunctions.c-list.simps*(*2*) *List.list.set-intros*(*1*))
**next**
**case** *False*
**then obtain** *x' xs'* **where** *x'-def*[*simp*]: *xs* = *x'*#*xs'* **using** *List.list.exhaust-sel* **by** *auto*
**then have** *c-list f cf ?h r* (*x* # *xs*)
= *c-list f cf ?h r xs* + *ldeep-T f cf xs* ∗ *c-list f cf ?h r* [*x*] **by** *simp*
**also have** ... = *c-list f cf ?h' r xs* + *ldeep-T f cf xs* ∗ *c-list f cf ?h r* [*x*]
**using** *Cons.IH*[*of ys*@[*x*]] *0* **by** *simp*
**also have** ... = *c-list f cf ?h'' r xs* + *ldeep-T f cf xs* ∗ *c-list f cf ?h r* [*x*]
**using** *Cons.IH*[*of* [*x*]] *1* **by** *simp*
**also have** ... = *c-list f cf ?h'' r xs* + *ldeep-T f cf xs* ∗ *c-list f cf ?h'' r* [*x*]
**using** *c-list-single-h-list-not-elem-prepend 2* **by** *metis*
**finally show** *?thesis* **by** *simp*
**qed**
**qed**(*simp*)

**lemma** *c-list-prepend-f-disjunct*:
**assumes** *distinct* (*ys*@*xs*)
**shows** *c-list* (*ldeep-s f* (*ys*@*xs*)) *cf h r xs* = *c-list* (*ldeep-s f xs*) *cf h r xs*

**using** *assms* **proof**(*induction xs arbitrary*: *ys*)
**case** (*Cons x xs*)
**then have** *0*: *distinct*(*ys* @ [*x*] @ *xs*) **by** *simp*
**then have** *1*: *distinct* ([*x*] @ *xs*) **by** *simp*
**let** *?f* = *ldeep-s f* (*ys* @ *x* # *xs*)
**let** *?f'* = *ldeep-s f xs*
**let** *?f''* = *ldeep-s f* (*x*#*xs*)
**have** *2*: *x* ∉ *set ys* **using** *Cons.prems* **by** *simp*
**show** *?case*
**proof**(*cases xs*=[])
**case** *False*
**then obtain** *x' xs'* **where** *x'-def*[*simp*]: *xs* = *x'*#*xs'* **using** *List.list.exhaust-sel*
**by** *auto*
**have** *ldeep-T ?f cf xs* = *ldeep-T ?f' cf xs*
**using** *distinct-ldeep-T-prepend*[*of ys*@[*x*] *xs f cf*] *Cons.prems* **by** *simp*
**then have** *3*: *ldeep-T ?f cf xs* = *ldeep-T ?f'' cf xs*
**using** *distinct-ldeep-T-prepend*[*of* [*x*] *xs f cf*] *Cons.prems 1* **by** *simp*
**have** *c-list ?f cf h r* (*x* # *xs*)
= *c-list ?f cf h r xs* + *ldeep-T ?f cf xs* ∗ *c-list ?f cf h r* [*x*]
**by** *simp*
**also have** ... = *c-list ?f' cf h r xs* + *ldeep-T ?f'' cf xs* ∗ *c-list ?f cf h r* [*x*]
**using** *Cons.IH*[*of ys*@[*x*]] *0 3* **by** *simp*
**also have** ... = *c-list ?f'' cf h r xs* + *ldeep-T ?f'' cf xs* ∗ *c-list ?f cf h r* [*x*]
**using** *Cons.IH*[*of* [*x*]] *1* **by** *simp*
**also have** ... = *c-list ?f'' cf h r xs* + *ldeep-T ?f'' cf xs* ∗ *c-list ?f'' cf h r* [*x*]
**using** *c-list-single-f-list-not-elem-prepend 2* **by** *metis*
**finally show** *?thesis* **by** *simp*
**qed**(*simp*)
**qed**(*simp*)

**lemma** *c-list'-eq-c-list*:
**assumes** *distinct xs*
**and** *rev xs* = *r* # *rs*
**shows** *c-list* (*ldeep-s f xs*) *cf* (*create-h-list h xs*) *r xs* = *c-list' f cf h xs*
**using** *assms* **proof**(*induction xs arbitrary*: *rs*)
**case** (*Cons x xs*)
**then show** *?case*
**proof**(*cases xs*=[])
**case** *False*
**then obtain** *x' xs'* **where** *x'-def*[*simp*]: *xs* = *x'*#*xs'* **using** *List.list.exhaust-sel*
**by** *auto*
**then have** *0*: *x* ≠ *r* **using** *distinct-first-uneq-last Cons* **by** *fast*
**have** *1*: *distinct xs* **using** *Cons.prems*(*1*) **by** *simp*
**have** ∃ *rs'*. *rev xs* = *r* # *rs'*
**using** *Cons.prems Nil-is-append-conv butlast-append*
**by** (*metis List.append.right-neutral List.butlast.simps*(*2*) *List.list.distinct*(*1*)
*List.rev.simps*(*2*) ‹⋀*thesis*. (⋀*x' xs'*. *xs* = *x'* # *xs'* ⟹ *thesis*) ⟹ *thesis*›)
**then obtain** *rs'* **where** *2*: *rev xs* = *r* # *rs'* **by** *blast*
**let** *?h* = *create-h-list h* (*x* # *x'* # *xs'*)

**let** *?h′ = create-h-list h (x′ # xs′)*
**let** *?f = ldeep-s f (x′#xs′)*
**let** *?f′ = ldeep-s f (x#x′#xs′)*
**have** *c-list (ldeep-s f (x#xs)) cf (create-h-list h (x # xs)) r (x # xs)*
*= c-list ?f′ cf ?h r (x # x′ # xs′)*
**by** *simp*
**also have** *... = c-list ?f′ cf ?h r (x′ # xs′)*
*+ ldeep-T ?f′ cf (x′ # xs′) ∗ c-list ?f′ cf ?h r [x]*
**by** *simp*
**also have** *... = c-list ?f′ cf ?h r (x′ # xs′) + ldeep-T ?f′ cf (x′ # xs′) ∗ h (x′ # xs′) x*
**using** *0* **by** *simp*
**also have** *... = c-list ?f′ cf ?h r (x′ # xs′) + ldeep-T ?f cf (x′ # xs′) ∗ h (x′ # xs′) x*
**using** *distinct-ldeep-T-prepend[of [x] x′#xs′] Cons.prems(1)* **by** *simp*
**also have** *... = c-list ?f′ cf ?h r (x′ # xs′) + ldeep-n f cf (x′ # xs′) ∗ h (x′ # xs′) x*
**using** *ldeep-T-eq-ldeep-n 1* **by** *fastforce*
**also have** *... = c-list ?f cf ?h r (x′ # xs′) + ldeep-n f cf (x′ # xs′) ∗ h (x′ # xs′) x*
**using** *c-list-prepend-f-disjunct[of [x] x′#xs′] Cons.prems(1)* **by** *simp*
**also have** *... = c-list ?f cf ?h′ r (x′ # xs′) + ldeep-n f cf (x′ # xs′) ∗ h (x′ # xs′) x*
**using** *c-list-prepend-h-disjunct[of [x] x′#xs′] Cons.prems* **by** *simp*
**also have** *... = c-list′ f cf h (x′ # xs′) + ldeep-n f cf (x′ # xs′) ∗ h (x′ # xs′) x*
**using** *Cons.IH 1 2* **by** *simp*
**also have** *... = c-list′ f cf h (x#x′ # xs′)*
**using** *Cons.prems x′-def 1 2* **by** *simp*
**finally show** *?thesis* **by** *simp*
**qed**(*simp*)
**qed**(*simp*)

**lemma** *clist-eq-if-cf-eq*:
*∀ x. set x ⊆ set xs ⟶ ldeep-T sf cf′ x = ldeep-T sf cf x*
*⟹ c-list sf cf′ h r xs = c-list sf cf h r xs*
**by** (*induction sf cf′ h r xs rule*: *c-list.induct*) (*auto simp*: *subset-insertI2*)

**lemma** *ldeep-s-h-eq-list-sel-aux′-h*:
*⟦distinct xs*; *ys@x#zs = xs⟧*
*⟹ (λa. ldeep-s f xs a ∗ cf a) x = (λxs x. (list-sel-aux′ f xs x) ∗ cf x) zs x*
**by** (*fastforce simp*: *distinct-ldeep-s-eq-aux*)

**corollary** *ldeep-s-h-eq-list-sel-aux′-h′*:
*⟦distinct-relations t*; *ys@x#zs = revorder t⟧*
*⟹ (λa. ldeep-s f (revorder t) a ∗ cf a) x = (λxs x. (list-sel-aux′ f xs x) ∗ cf x) zs x*
**by** (*fastforce simp*: *distinct-rels-alt ldeep-s-h-eq-list-sel-aux′-h*)

**lemma** *create-h-list-distinct-simp*: ⟦*distinct xs*; *ys*@*x*#*zs* = *xs*⟧ ⟹ *create-h-list h xs x* = *h zs x*
**by** (*induction xs arbitrary*: *ys*) (*force simp*: *append-eq-Cons-conv*)+

**lemma** *ldeep-s-h-eq-create-h-list*:
⟦*distinct xs*; *ys*@*x*#*zs* = *xs*⟧
⟹ (λ*a*. *ldeep-s f xs a* ∗ *cf a*) *x* = *create-h-list* (λ*xs x*. (*list-sel-aux*′ *f xs x*) ∗ *cf x*) *xs x*
**by** (*simp add*: *distinct-relations-def create-h-list-distinct-simp ldeep-s-h-eq-list-sel-aux*′*-h*)

**lemma** *ldeep-s-h-eq-create-h-list*′:
⟦*distinct-relations t*; *ys*@*x*#*zs* = *revorder t*⟧
⟹ (λ*a*. *ldeep-s f* (*revorder t*) *a* ∗ *cf a*) *x*
= *create-h-list* (λ*xs x*. (*list-sel-aux*′ *f xs x*) ∗ *cf x*) (*revorder t*) *x*
**by** (*simp add*: *distinct-rels-alt ldeep-s-h-eq-create-h-list*)

**corollary** *ldeep-s-h-eq-create-h-list*′′:
*distinct-relations t* ⟹ ∀ *ys x zs*. *ys*@*x*#*zs* = *revorder t*
⟶ (λ*a*. *ldeep-s f* (*revorder t*) *a* ∗ *cf a*) *x*
= *create-h-list* (λ*xs x*. (*list-sel-aux*′ *f xs x*) ∗ *cf x*) (*revorder t*) *x*
**using** *ldeep-s-h-eq-create-h-list*′ **by** *fast*

**lemma** *ldeep-s-h-eq-create-h-list*′′′:
⟦*distinct-relations t*; *x* ∈ *relations t*⟧
⟹ (λ*a*. *ldeep-s f* (*revorder t*) *a* ∗ *cf a*) *x*
= *create-h-list* (λ*xs x*. (*list-sel-aux*′ *f xs x*) ∗ *cf x*) (*revorder t*) *x*
**using** *ldeep-s-eq-list-sel-aux*′*-split revorder-eq-set*
**by** (*fastforce simp add*: *distinct-rels-alt ldeep-s-h-eq-create-h-list*)

**lemma** *cons2-if-2elems*: ⟦*x* ∈ *set xs*; *y* ∈ *set xs*; *x* ≠ *y*⟧ ⟹ ∃ *y z zs*. *xs* = *y* # *z* # *zs*
**using** *last.simps list.set-cases neq-Nil-conv* **by** *metis*

**theorem** *c-IKKBZ-eq-c-list*:
**fixes** *t*
**defines** *xs* ≡ *revorder t*
**assumes** *distinct-relations t*
**and** *reasonable-cards cf f t*
**and** *left-deep t*
**and** ∀ *x* ∈ *relations t*. *h1 x* (*cf x*) = *h2 x*
**shows** *c-IKKBZ h1 cf f t* = *c-list* (*ldeep-s f xs*) *cf h2* (*first-node t*) *xs*
**using** *assms* **proof**(*induction t arbitrary*: *xs rule*: *left-deep.induct*)
**case** (*2 l r*)
**let** *?r* = *first-node* (*Join l* (*Relation r*))
**let** *?xs* = *revorder* (*Join l* (*Relation r*))
**let** *?ys* = *revorder l*
**let** *?sf* = *ldeep-s f ?xs*
**have** *h1-h2-l*: ∀ *x* ∈ *relations l*. *h1 x* (*cf x*) = *h2 x* **using** *2.prems*(*4*) **by** *simp*
**have** *c-IKKBZ h1 cf f* (*Join l* (*Relation r*)) = *card cf f l* ∗ (*h1 r* (*cf r*)) +

*c-IKKBZ h1 cf f l*
**by** *simp*
**then have** *c-IKKBZ h1 cf f (Join l (Relation r))*
= *card cf f l ∗ (h1 r (cf r)) + c-list (ldeep-s f ?ys) cf h2 ?r ?ys*
**using** *2.hyps 2.prems(2−3) distinct-trans-l[OF 2.prems(1)] h1-h2-l* **by** *force*
**then have** *ind: c-IKKBZ h1 cf f (Join l (Relation r))*
= *card cf f l ∗ (h1 r (cf r)) + c-list ?sf cf h2 ?r ?ys*
**using** *c-list-prepend-f-disjunct 2.prems(1)* **unfolding** *distinct-rels-alt*
**by** (*metis revorder.simps(2)*)
**have** *0: ?r ∈ set ?xs* **using** *first-node-last-revorder[of l]* **by** *force*
**moreover have** *1: r ∈ set ?xs* **by** *simp*
**moreover have** *distinct ?xs* **using** *2.prems(1) distinct-rels-alt* **by** *force*
**ultimately have** *?r ≠ r* **using** *first-node-last-revorder[of l]* **by** *auto*
**then obtain** *z zs* **where** *z-def: ?xs = r # z # zs* **using** *cons2-if-2elems[OF 0 1]* **by** *auto*
**then have** *c-list ?sf cf h2 ?r ?xs*
= *c-list ?sf cf h2 ?r ?ys + ldeep-T ?sf cf ?ys ∗ c-list ?sf cf h2 ?r [r]*
**by** *simp*
**also have** *... = c-list ?sf cf h2 ?r ?ys + ldeep-T ?sf cf ?ys ∗ (h1 r (cf r))*
**using** ‹*?r ≠ r*› *2.prems(4)* **by** *fastforce*
**also have** *... = c-list ?sf cf h2 ?r ?ys + card cf f l ∗ (h1 r (cf r))*
**using** *2.prems(1,3) ldeep-T-eq-card distinct-rels-alt distinct-ldeep-T-prepend*
**by** (*metis revorder.simps(2) ldeep-trans distinct-trans-l*)
**finally show** *?case* **using** *ind* **by** *simp*
**qed**(*auto*)

**lemma** *c-IKKBZ-eq-c-list-cout*:
**fixes** *f cf t*
**defines** *xs ≡ revorder t*
**defines** *h ≡ (λa. ldeep-s f xs a ∗ cf a)*
**assumes** *distinct-relations t*
**and** *reasonable-cards cf f t*
**and** *left-deep t*
**shows** *c-IKKBZ (λa b. ldeep-s f xs a ∗ b) cf f t = c-list (ldeep-s f xs) cf h (first-node t) xs*
**using** *assms c-IKKBZ-eq-c-list* **by** *fast*

**lemma** *c-IKKBZ-eq-c-list-cout-hlist*:
**fixes** *f cf t*
**defines** *h ≡ (λxs x. (list-sel-aux′ f xs x) ∗ cf x)*
**defines** *xs ≡ revorder t*
**assumes** *distinct-relations t*
**and** *reasonable-cards cf f t*
**and** *left-deep t*
**shows** *c-IKKBZ (λa b. ldeep-s f xs a ∗ b) cf f t*
= *c-list (ldeep-s f xs) cf (create-h-list h xs) (first-node t) xs*
**using** *assms c-IKKBZ-eq-c-list ldeep-s-h-eq-create-h-list′′′[OF assms(3)]* **by** *fastforce*

**theorem** *c-out-eq-c-list*:
**fixes** *f cf t*
**defines** *xs* ≡ *revorder t*
**defines** *h* ≡ (λ*a*. *ldeep-s f xs a* ∗ *cf a*)
**assumes** *distinct-relations t*
**and** *reasonable-cards cf f t*
**and** *left-deep t*
**shows** *c-list* (*ldeep-s f xs*) *cf h* (*first-node t*) *xs* = *c-out cf f t*
**using** *c-IKKBZ-eq-c-list-cout c-out-IKKBZ assms* **by** *fastforce*

**theorem** *c-out-eq-c-list-hlist*:
**fixes** *f cf t*
**defines** *h* ≡ (λ*xs x*. (*list-sel-aux′ f xs x*) ∗ *cf x*)
**defines** *xs* ≡ *revorder t*
**assumes** *distinct-relations t*
**and** *reasonable-cards cf f t*
**and** *left-deep t*
**shows** *c-list* (*ldeep-s f xs*) *cf* (*create-h-list h xs*) (*first-node t*) *xs* = *c-out cf f t*
**using** *c-IKKBZ-eq-c-list-cout-hlist c-out-IKKBZ assms* **by** *fastforce*

**lemma** *c-out-eq-c-list-altproof*:
**fixes** *f cf t*
**defines** *h* ≡ (λ*xs x*. (*list-sel-aux′ f xs x*) ∗ *cf x*)
**defines** *xs* ≡ *revorder t*
**assumes** *distinct-relations t*
**and** *reasonable-cards cf f t*
**and** *left-deep t*
**shows** *c-list* (*ldeep-s f xs*) *cf* (*create-h-list h xs*) (*first-node t*) *xs* = *c-out cf f t*
**proof** −
**obtain** *rs* **where** *rs-def*[*simp*]: *rev* (*revorder t*) = (*first-node t*) # *rs*
**unfolding** *revorder-eq-rev-inorder* **using** *first-node-first-inorder* **by** *auto*
**have** *0*: *distinct* (*revorder t*) **using** *assms*(*3*) *distinct-rels-alt* **by** *auto*
**then have** *c-list* (*ldeep-s f xs*) *cf* (*create-h-list h xs*) (*first-node t*) *xs*
= *c-list′ f cf h* (*revorder t*)
**using** *rs-def c-list′-eq-c-list xs-def* **by** *fast*
**then show** *?thesis* **using** *assms c-out-eq-c-list′* **by** *auto*
**qed**

Similarly, we can derive the equivalence for other cost functions like *c-nlj* and *c-hj* by using the equivalence of *c-IKKBZ* and *c-list*.

**lemma** *c-IKKBZ-eq-c-list-hj*:
**fixes** *f cf t*
**defines** *xs* ≡ *revorder t*
**assumes** *distinct-relations t*
**and** *reasonable-cards cf f t*
**and** *left-deep t*
**shows** *c-IKKBZ* (λ- -. *1.2*) *cf f t* = *c-list* (*ldeep-s f xs*) *cf* (λ-. *1.2*) (*first-node t*) *xs*

**using** *c-IKKBZ-eq-c-list assms* **by** *fast*

**corollary** *c-hj-eq-c-list*:
**fixes** *f cf t*
**defines** *xs* ≡ *revorder t*
**assumes** *distinct-relations t*
**and** *reasonable-cards cf f t*
**and** *left-deep t*
**shows** *c-list* (*ldeep-s f xs*) *cf* (λ-. *1.2*) (*first-node t*) *xs* = *c-hj cf f t*
**using** *c-IKKBZ-eq-c-list-hj c-hj-IKKBZ assms* **by** *fastforce*

**lemma** *c-IKKBZ-eq-c-list-nlj*:
**fixes** *f cf t*
**defines** *xs* ≡ *revorder t*
**assumes** *distinct-relations t*
**and** *reasonable-cards cf f t*
**and** *left-deep t*
**shows** *c-IKKBZ* (λ-. *id*) *cf f t* = *c-list* (*ldeep-s f xs*) *cf cf* (*first-node t*) *xs*
**using** *c-IKKBZ-eq-c-list assms* **by** *fastforce*

**corollary** *c-nlj-eq-c-list*:
**fixes** *f cf t*
**defines** *xs* ≡ *revorder t*
**assumes** *distinct-relations t*
**and** *reasonable-cards cf f t*
**and** *left-deep t*
**shows** *c-list* (*ldeep-s f xs*) *cf cf* (*first-node t*) *xs* = *c-nlj cf f t*
**using** *c-IKKBZ-eq-c-list-nlj c-nlj-IKKBZ assms* **by** *fastforce*

**lemma** *c-list-app*:
*c-list f cf h r* (*ys*@*xs*) = *c-list f cf h r xs* + *ldeep-T f cf xs* ∗ *c-list f cf h r ys*
**proof**(*induction ys*)
**case** (*Cons y ys*)
**then show** *?case*
**proof**(*cases xs*=[])
**case** *True*
**then show** *?thesis* **using** *ldeep-T-empty* **by** *auto*
**next**
**case** *False*
**then obtain** *x′ xs′* **where** *x′-def*[*simp*]: *xs* = *x′*#*xs′* **using** *List.list.exhaust-sel*
**by** *blast*
**then have** *c-list f cf h r* (*y*#*ys* @ *xs*)
= *c-list f cf h r* (*ys*@*xs*) + *ldeep-T f cf* (*ys*@*xs*) ∗ *c-list f cf h r* [*y*]
**by** (*metis CostFunctions.c-list.simps*(*3*) *Nil-is-append-conv neq-Nil-conv*)
**also have** ... = *c-list f cf h r xs* + *ldeep-T f cf xs* ∗ *c-list f cf h r ys*
+ *ldeep-T f cf* (*ys*@*xs*) ∗ *c-list f cf h r* [*y*]
**using** *Cons.IH* **by** *simp*
**also have** ... = *c-list f cf h r xs* + *ldeep-T f cf xs* ∗ *c-list f cf h r ys*
+ *ldeep-T f cf ys* ∗ *ldeep-T f cf xs* ∗ *c-list f cf h r* [*y*]

```
    using ldeep-T-app by auto
  also have ... = c-list f cf h r xs + ldeep-T f cf xs * (c-list f cf h r ys
        + ldeep-T f cf ys * c-list f cf h r [y])
    by argo
  also have ... = c-list f cf h r xs + ldeep-T f cf xs * (c-list f cf h r (y # ys))
    using False neq-Nil-conv List.append.left-neutral
    by (metis CostFunctions.c-list.simps(3) calculation)
  finally show ?thesis by simp
 qed
qed(simp)

lemma create-h-list-pos:
 ⟦sel-reasonable sf; ∀ x ∈ set xs. cf x > 0⟧
  ⟹ (create-h-list (λxs x. (list-sel-aux' sf xs x) * cf x) xs) x > 0
 by (induction xs) (auto simp: list-sel-aux'-reasonable)

lemma c-list-not-neg:
 assumes sel-reasonable sf
    and ∀ x ∈ set ys. cf x > 0
    and h = (λa. ldeep-s sf xs a * cf a)
  shows c-list (ldeep-s sf xs) cf h r ys ≥ 0
using assms proof(induction ys arbitrary: xs)
 case ind: (Cons y ys)
 let ?sf = ldeep-s sf xs
 show ?case
 proof(cases ys)
  case Nil
  then show ?thesis using ind.prems by (simp add: ldeep-s-pos order-less-imp-le)
 next
  case (Cons y' ys')
  show ?thesis
  proof(cases y=r)
   case True
   then show ?thesis using Cons ind by simp
  next
   case False
   have c-list ?sf cf h r (y # ys) = c-list ?sf cf h r ys + ldeep-T ?sf cf ys * h y
    using Cons False by simp
   then have c-list ?sf cf h r (y # ys) ≥ ldeep-T ?sf cf ys * h y
    using ind by simp
   moreover have ldeep-T ?sf cf ys * h y > 0
    using ind.prems by (simp add: ldeep-T-pos ldeep-s-pos)
   ultimately show ?thesis by simp
  qed
 qed
qed(simp)

lemma c-list-not-neg-hlist:
 assumes sel-reasonable sf
```

```
      and ∀ x ∈ set xs. cf x > 0
      and ∀ x ∈ set ys. cf x > 0
      and h = create-h-list (λxs x. (list-sel-aux' sf xs x) * cf x) xs
    shows c-list (ldeep-s sf xs) cf h r ys ≥ 0
using assms proof(induction ys arbitrary: xs)
  case ind: (Cons y ys)
  let ?sf = ldeep-s sf xs
  show ?case
  proof(cases ys)
    case Nil
    then show ?thesis
      using ind.prems by(cases y=r)(auto simp: create-h-list-pos less-eq-real-def)
  next
    case (Cons y' ys')
    show ?thesis
    proof(cases y=r)
      case True
      then show ?thesis using Cons ind by simp
    next
      case False
      have c-list ?sf cf h r (y # ys) = c-list ?sf cf h r ys + ldeep-T ?sf cf ys * h y
        using Cons False by simp
      then have c-list ?sf cf h r (y # ys) ≥ ldeep-T ?sf cf ys * h y
        using ind by simp
      moreover have ldeep-T ?sf cf ys * h y > 0
        using create-h-list-pos[of sf xs cf y] ind.prems by (simp add: ldeep-T-pos)
      ultimately show ?thesis by simp
    qed
  qed
qed(simp)

lemma c-list-pos-if-h-pos:
  ⟦sel-reasonable sf; ∀ x ∈ set xs. cf x > 0; ∀ x ∈ set xs. h x > 0; r ∉ set xs; xs ≠
[]⟧
    ⟹ c-list (ldeep-s sf ys) cf h r xs > 0
proof(induction ldeep-s sf ys cf h r xs rule: c-list.induct)
  case (3 cf h r y x xs)
 have ldeep-T (ldeep-s sf ys) cf (x#xs) > 0 using ldeep-T-pos[of x#xs] 3.prems(1,2)
by simp
  then have ldeep-T (ldeep-s sf ys) cf (x#xs) * c-list (ldeep-s sf ys) cf h r [y] > 0
    using 3 by auto
  moreover have c-list (ldeep-s sf ys) cf h r (x#xs) > 0 using 3 by auto
  ultimately show ?case by simp
qed(auto)

lemma c-list-pos-r-not-elem:
  assumes sel-reasonable sf
      and ∀ x ∈ set ys. cf x > 0
      and ys ≠ []
```

```
    and r ∉ set ys
    and h = (λa. ldeep-s sf xs a ∗ cf a)
  shows c-list (ldeep-s sf xs) cf h r ys > 0
 using c-list-pos-if-h-pos ldeep-s-pos assms by fastforce

lemma c-list-pos-r-not-elem-hlist:
 assumes sel-reasonable sf
    and ∀ x ∈ set xs. cf x > 0
    and ∀ x ∈ set ys. cf x > 0
    and ys ≠ []
    and r ∉ set ys
    and h = create-h-list (λxs x. (list-sel-aux' sf xs x) ∗ cf x) xs
  shows c-list (ldeep-s sf xs) cf h r ys > 0
 using c-list-pos-if-h-pos create-h-list-pos[OF assms(1)] assms by fastforce

lemma c-list-pos-not-root:
 assumes sel-reasonable sf
    and ∀ x ∈ set ys. cf x > 0
    and ys ≠ []
    and ys ≠ [r]
    and distinct ys
    and h = (λa. ldeep-s sf xs a ∗ cf a)
  shows c-list (ldeep-s sf xs) cf h r ys > 0
using assms proof(induction ys arbitrary: xs)
 case ind: (Cons y ys)
 let ?sf = ldeep-s sf xs
 show ?case
 proof(cases ys)
  case Nil
  then have c-list ?sf cf h r (y # ys) = h y using ind.prems(4) by simp
  then show ?thesis using ind.prems(1,2,6) by (simp add: ldeep-s-pos)
 next
  case (Cons y' ys')
  show ?thesis
  proof(cases y=r)
   case True
   then have 0: r ∉ set ys using ind.prems(5) by simp
   have c-list ?sf cf h r (y # ys) = c-list ?sf cf h r ys
    using Cons True by simp
   then show ?thesis using ind.prems(1,2,4,6) 0 True by (fastforce intro:
c-list-pos-r-not-elem)
  next
   case False
   have c-list ?sf cf h r (y # ys) = c-list ?sf cf h r ys + ldeep-T ?sf cf ys ∗ h y
    using Cons False by simp
   then have c-list ?sf cf h r (y # ys) ≥ ldeep-T ?sf cf ys ∗ h y
    using c-list-not-neg ind.prems(1,2,3,6) by fastforce
   moreover have ldeep-T ?sf cf ys ∗ h y > 0
    using ind.prems(1,2,6) by (simp add: ldeep-T-pos ldeep-s-pos)
```

```
    ultimately show ?thesis by simp
   qed
  qed
qed(simp)

lemma c-list-pos-not-root-hlist:
  assumes sel-reasonable sf
      and ∀ x ∈ set xs. cf x > 0
      and ∀ x ∈ set ys. cf x > 0
      and ys ≠ []
      and ys ≠ [r]
      and distinct ys
      and h = create-h-list (λxs x. (list-sel-aux' sf xs x) * cf x) xs
    shows c-list (ldeep-s sf xs) cf h r ys > 0
using assms proof(induction ys arbitrary: xs)
  case ind: (Cons y ys)
  let ?sf = ldeep-s sf xs
  show ?case
  proof(cases ys)
    case Nil
    then have c-list ?sf cf h r (y # ys) = h y using ind.prems(5) by simp
    then show ?thesis using create-h-list-pos ind.prems(1,2,7) by fastforce
  next
    case (Cons y' ys')
    show ?thesis
    proof(cases y=r)
      case True
      then have 0: r ∉ set ys using ind.prems(6) by simp
      have c-list ?sf cf h r (y # ys) = c-list ?sf cf h r ys
        using Cons True by simp
      then show ?thesis
       using c-list-pos-r-not-elem-hlist[of sf xs cf ys r h] 0 ind.prems(1,2,3,7) Cons
by auto
    next
      case False
      have c-list ?sf cf h r (y # ys) = c-list ?sf cf h r ys + ldeep-T ?sf cf ys * h y
        using Cons False by simp
      then have c-list ?sf cf h r (y # ys) ≥ ldeep-T ?sf cf ys * h y
        using c-list-not-neg-hlist ind.prems(1,2,3,7) by fastforce
      moreover have ldeep-T ?sf cf ys * h y > 0
        using ind.prems(1,2,3,7) by (simp add: ldeep-T-pos create-h-list-pos)
      ultimately show ?thesis by simp
    qed
  qed
qed(simp)

lemma c-list-split-four:
  assumes T = ldeep-T f cf
      and C = c-list f cf h r
```

```
  shows C (rev (A @ U @ V @ B)) = C (rev A) + T (rev A) * C (rev U)
        + T (rev A) * T (rev U) * C (rev V)
        + T (rev A) * T (rev U) * T (rev V) * C (rev B)
proof −
  let ?T = ldeep-T f cf
  let ?C = c-list f cf h r
  have ?C (rev (A @ U @ V @ B))
    = ?C (rev A) + ?T (rev A) * ?C (rev (U @ V @ B))
    using c-list-app[where ys=rev (U@V@B)] by simp
  also have ... = ?C (rev A) + ?T (rev A) * (?C (rev U)
      + ?T (rev U) * ?C (rev (V@B)))
    using c-list-app[where ys=rev (V@B)] by simp
  also have ... = ?C (rev A) + ?T (rev A) * ?C (rev U)
      + ?T (rev A) * ?T (rev U) * ?C (rev (V@B))
    by argo
  also have ... = ?C (rev A) + ?T (rev A) * ?C (rev U)
      + ?T (rev A) * ?T (rev U) * (?C (rev V)
      + ?T (rev V) * ?C (rev B))
    using c-list-app by force
  finally have 0: ?C (rev (A @ U @ V @ B))
    = ?C (rev A) + ?T (rev A) * ?C (rev U)
      + ?T (rev A) * ?T (rev U) * ?C (rev V)
      + ?T (rev A) * ?T (rev U) * ?T (rev V) * ?C (rev B)
    by argo
  then show ?thesis using assms by simp
qed

lemma c-list-A-pos-asi:
  assumes c-list f cf h r (rev U) > 0
    and c-list f cf h r (rev V) > 0
    and ldeep-T f cf (rev A) > 0
  shows c-list f cf h r (rev (A @ U @ V @ B)) ≤ c-list f cf h r (rev (A @ V @
U @ B))
    ⟷ ((ldeep-T f cf (rev U) − 1) / c-list f cf h r (rev U)
      ≤ (ldeep-T f cf (rev V) − 1) / c-list f cf h r (rev V))
proof −
  let ?T = ldeep-T f cf
  let ?C = c-list f cf h r
  let ?rank = (λl. (?T l − 1) / ?C l)
  have 0: ?C (rev (A @ U @ V @ B))
    = ?C (rev A) + ?T (rev A) * ?C (rev U)
      + ?T (rev A) * ?T (rev U) * ?C (rev V)
      + ?T (rev A) * ?T (rev U) * ?T (rev V) * ?C (rev B)
    using c-list-split-four by fastforce
  have ?C (rev (A @ V @ U @ B))
    = ?C (rev A) + ?T (rev A) * ?C (rev V)
      + ?T (rev A) * ?T (rev V) * ?C (rev U)
      + ?T (rev A) * ?T (rev V) * ?T (rev U) * ?C (rev B)
    using c-list-split-four by fastforce
```

```
  then have ?C (rev (A@U@V@B)) − ?C (rev (A@V@U@B))
        = ?T (rev A) ∗ (?C (rev V) ∗ (?T (rev U) − 1) − ?C (rev U) ∗ (?T (rev
V) − 1))
    using 0 by argo
  also have ... = ?T (rev A) ∗
            (?C (rev V) ∗ (?T (rev U) − 1) ∗ (?C (rev U) / ?C (rev U))
              − ?C (rev U) ∗ (?T (rev V) − 1) ∗ (?C (rev V) / ?C (rev V)))
    using assms
   by (metis Groups.monoid-mult-class.mult.right-neutral divide-self-if less-numeral-extra(3))
  also have ... = ?T (rev A) ∗ ?C (rev U) ∗ ?C (rev V) ∗ (?rank (rev U) −
?rank (rev V))
    by argo
  finally have 1: ?C (rev (A@U@V@B)) − ?C (rev (A@V@U@B))
            = ?T (rev A) ∗ ?C (rev U) ∗ ?C (rev V) ∗ (?rank (rev U) − ?rank
(rev V)).
  then show ?thesis
  proof(cases ?C (rev (A@U@V@B)) ≤ ?C (rev (A@V@U@B)))
    case True
    then show ?thesis by (smt (verit) assms 1 mult-pos-pos)
  next
    case False
    then show ?thesis by (smt (z3) 1 assms mult-pos-pos zero-less-mult-pos)
  qed
qed

lemma c-list-asi-aux:
  assumes sel-reasonable sf
      and ∀ x. cf x > 0
      and c = c-list f cf h r
      and f = (ldeep-s sf xs)
      and ∀ ys. (ys ≠ [] ∧ r ∉ set ys) ⟶ c ys > 0
      and distinct (A@U@V@B)
      and U ≠ []
      and V ≠ []
      and rank = (λl. (ldeep-T f cf l − 1) / c l)
    and r ∉ set (A@U@V@B) ∨ (take 1 (A@U@V@B) = [r] ∧ take 1 (A@V@U@B)
= [r])
    shows (c (rev (A@U@V@B)) ≤ c (rev (A@V@U@B)) ⟷ rank (rev U) ≤
rank (rev V))
proof (cases r ∉ set (A@U@V@B))
  case True
  have 0: ldeep-T f cf (rev A) > 0 using assms(1,2,4) ldeep-T-pos by fast
  have r ∉ set (rev U) using True by simp
  then have 1: c-list f cf h r (rev U) > 0
    using c-list-pos-r-not-elem assms(1−5,7) by fastforce
  have r ∉ set (rev V) using True by simp
  then have c-list f cf h r (rev V) > 0
    using c-list-pos-r-not-elem assms(1−5,8) by fastforce
  then show ?thesis using c-list-A-pos-asi 0 1 assms(3,9) by fast
```

**next**
**case** *False*
**have** *0*: *ldeep-T f cf* (*rev A*) > *0* **using** *assms*(*1*,*2*,*4*) *ldeep-T-pos* **by** *fast*
**have** *r-first*: *take 1* (*A*@*U*@*V*@*B*) = [*r*] ∧ *take 1* (*A*@*V*@*U*@*B*) = [*r*]
**using** *assms*(*10*) *False* **by** *blast*
**then have** *take 1 A* = [*r*] **using** *assms*(*6*−*8*) *distinct-change-order-first-elem* **by** *metis*
**then have** *r* ∈ *set A* **by** (*metis List.list.set-intros*(*1*) *in-set-takeD*)
**then have** *1*: *r* ∉ *set* (*U*@*V*@*B*) **using** *assms*(*6*) **by** *auto*
**then have** *r* ∉ *set* (*rev U*) **by** *simp*
**then have** *2*: *c-list f cf h r* (*rev U*) > *0*
**using** *c-list-pos-r-not-elem assms*(*1*−*5*,*7*) **by** *fastforce*
**have** *r* ∉ *set* (*rev V*) **using** *1* **by** *simp*
**then have** *c-list f cf h r* (*rev V*) > *0*
**using** *c-list-pos-r-not-elem assms*(*1*−*5*,*8*) **by** *fastforce*
**then show** *?thesis* **using** *c-list-A-pos-asi 0 2 assms*(*3*,*9*) **by** *fast*
**qed**

**lemma** *c-list-pos-asi*:
**fixes** *sf cf h r xs*
**defines** *f* ≡ *ldeep-s sf xs*
**defines** *rank* ≡ (λ*l*. (*ldeep-T f cf l* − *1*) / *c-list f cf h r l*)
**assumes** *sel-reasonable sf*
**and** ∀ *x*. *cf x* > *0*
**and** ∀ *ys*. (*ys* ≠ [] ∧ *r* ∉ *set ys*) ⟶ *c-list f cf h r ys* > *0*
**shows** *asi rank r* (*c-list f cf h r*)
**unfolding** *asi-def* **using** *c-list-asi-aux*[*OF assms*(*3*,*4*)] *assms*(*1*,*2*,*5*) **by** *simp*

**theorem** *c-list-asi*:
**fixes** *sf cf h r xs*
**defines** *f* ≡ *ldeep-s sf xs*
**defines** *rank* ≡ (λ*l*. (*ldeep-T f cf l* − *1*) / *c-list f cf h r l*)
**assumes** *sel-reasonable sf*
**and** ∀ *x*. *cf x* > *0*
**and** ∀ *x*. *h x* > *0*
**shows** *asi rank r* (*c-list f cf h r*)
**using** *c-list-pos-asi assms c-list-pos-if-h-pos*[*OF assms*(*3*)] **by** *fastforce*

**corollary** *c-out-asi*:
**fixes** *sf cf r xs*
**defines** *f* ≡ *ldeep-s sf xs*
**defines** *h* ≡ (λ*a*. *ldeep-s sf xs a* ∗ *cf a*)
**defines** *rank* ≡ (λ*l*. (*ldeep-T f cf l* − *1*) / *c-list f cf h r l*)
**assumes** *sel-reasonable sf*
**and** ∀ *x*. *cf x* > *0*
**shows** *asi rank r* (*c-list f cf h r*)
**using** *c-list-asi ldeep-s-pos assms* **by** *fastforce*

**lemma** *c-out-asi-aux*:
**assumes** *sel-reasonable sf*
**and** $\forall x.\ cf\ x > 0$
**and** $c = c\text{-}list\ f\ cf\ h\ r$
**and** $f = (ldeep\text{-}s\ sf\ xs)$
**and** $h = (\lambda a.\ ldeep\text{-}s\ sf\ xs\ a * cf\ a)$
**and** $distinct\ (A@U@V@B)$
**and** $U \neq []$
**and** $V \neq []$
**and** $rank = (\lambda l.\ (ldeep\text{-}T\ f\ cf\ l - 1)\ /\ c\ l)$
**and** $r \notin set\ (A@U@V@B) \vee (take\ 1\ (A@U@V@B) = [r] \wedge take\ 1\ (A@V@U@B) = [r])$
**shows** $(c\ (rev\ (A@U@V@B)) \leq c\ (rev\ (A@V@U@B)) \longleftrightarrow rank\ (rev\ U) \leq rank\ (rev\ V))$
**proof** $(cases\ r \notin set\ (A@U@V@B))$
**case** *True*
**have** *0*: $ldeep\text{-}T\ f\ cf\ (rev\ A) > 0$ **using** *assms*(*1*,*2*,*4*) *ldeep-T-pos* **by** *fast*
**have** $r \notin set\ (rev\ U)$ **using** *True* **by** *simp*
**then have** *1*: $c\text{-}list\ f\ cf\ h\ r\ (rev\ U) > 0$
**using** *c-list-pos-r-not-elem assms*(*1*,*2*,*4*,*5*,*7*) **by** *fastforce*
**have** $r \notin set\ (rev\ V)$ **using** *True* **by** *simp*
**then have** $c\text{-}list\ f\ cf\ h\ r\ (rev\ V) > 0$
**using** *c-list-pos-r-not-elem assms*(*1*,*2*,*4*,*5*,*8*) **by** *fastforce*
**then show** *?thesis* **using** *c-list-A-pos-asi 0 1 assms*(*3*,*9*) **by** *fast*
**next**
**case** *False*
**have** *0*: $ldeep\text{-}T\ f\ cf\ (rev\ A) > 0$ **using** *assms*(*1*,*2*,*4*) *ldeep-T-pos* **by** *fast*
**have** *r-first*: $take\ 1\ (A@U@V@B) = [r] \wedge take\ 1\ (A@V@U@B) = [r]$
**using** *assms*(*10*) *False* **by** *blast*
**then have** $take\ 1\ A = [r]$ **using** *assms*(*6*−*8*) *distinct-change-order-first-elem* **by** *metis*
**then have** $r \in set\ A$ **by** (*metis List.list.set-intros*(*1*) *in-set-takeD*)
**then have** *1*: $r \notin set\ (U@V@B)$ **using** *assms*(*6*) **by** *auto*
**then have** $r \notin set\ (rev\ U)$ **by** *simp*
**then have** *2*: $c\text{-}list\ f\ cf\ h\ r\ (rev\ U) > 0$
**using** *c-list-pos-r-not-elem assms*(*1*,*2*,*4*,*5*,*7*) **by** *fastforce*
**have** $r \notin set\ (rev\ V)$ **using** *1* **by** *simp*
**then have** $c\text{-}list\ f\ cf\ h\ r\ (rev\ V) > 0$
**using** *c-list-pos-r-not-elem assms*(*1*,*2*,*4*,*5*,*8*) **by** *fastforce*
**then show** *?thesis* **using** *c-list-A-pos-asi 0 2 assms*(*3*,*9*) **by** *fast*
**qed**

**lemma** *c-out-asi-aux-hlist*:
**assumes** *sel-reasonable sf*
**and** $\forall x.\ cf\ x > 0$
**and** $c = c\text{-}list\ f\ cf\ h\ r$
**and** $f = (ldeep\text{-}s\ sf\ xs)$
**and** $h = create\text{-}h\text{-}list\ (\lambda xs\ x.\ (list\text{-}sel\text{-}aux'\ sf\ xs\ x) * cf\ x)\ xs$
**and** $distinct\ (A@U@V@B)$

```
      and U ≠ []
      and V ≠ []
      and rank = (λl. (ldeep-T f cf l − 1) / c l)
    and r ∉ set (A@U@V@B) ∨ (take 1 (A@U@V@B) = [r] ∧ take 1 (A@V@U@B)
= [r])
    shows (c (rev (A@U@V@B)) ≤ c (rev (A@V@U@B)) ⟷ rank (rev U) ≤
rank (rev V))
proof (cases r ∉ set (A@U@V@B))
  case True
  have 0: ldeep-T f cf (rev A) > 0 using assms(1,2,4) ldeep-T-pos by fast
  have r ∉ set (rev U) using True by simp
  then have 1: c-list f cf h r (rev U) > 0
    using c-list-pos-r-not-elem-hlist assms(1,2,4,5,7) by fastforce
  have r ∉ set (rev V) using True by simp
  then have c-list f cf h r (rev V) > 0
    using c-list-pos-r-not-elem-hlist assms(1,2,4,5,8) by fastforce
  then show ?thesis using c-list-A-pos-asi 0 1 assms(3,9) by fast
next
  case False
  have 0: ldeep-T f cf (rev A) > 0 using assms(1,2,4) ldeep-T-pos by fast
  have r-first: take 1 (A@U@V@B) = [r] ∧ take 1 (A@V@U@B) = [r]
    using assms(10) False by blast
 then have take 1 A = [r] using assms(6−8) distinct-change-order-first-elem by
metis
  then have r ∈ set A by (metis List.list.set-intros(1) in-set-takeD)
  then have 1: r ∉ set (U@V@B) using assms(6) by auto
  then have r ∉ set (rev U) by simp
  then have 2: c-list f cf h r (rev U) > 0
    using c-list-pos-r-not-elem-hlist assms(1,2,4,5,7) by fastforce
  have r ∉ set (rev V) using 1 by simp
  then have c-list f cf h r (rev V) > 0
    using c-list-pos-r-not-elem-hlist assms(1,2,4,5,8) by fastforce
  then show ?thesis using c-list-A-pos-asi 0 2 assms(3,9) by fast
qed

theorem c-out-asi-altproof:
  assumes sel-reasonable sf
      and ∀x. cf x > 0
      and c = c-list f cf h r
      and f = (ldeep-s sf xs)
      and h = (λa. ldeep-s sf xs a ∗ cf a)
    shows asi (λl. (ldeep-T f cf l − 1) / c l) r (c-list f cf h r)
  unfolding asi-def using c-out-asi-aux[OF assms] assms(3) by blast

theorem c-out-asi-hlist:
  assumes sel-reasonable sf
      and ∀x. cf x > 0
      and c = c-list f cf h r
      and f = (ldeep-s sf xs)
```

**and** $h = create\text{-}h\text{-}list\ (\lambda xs\ x.\ (list\text{-}sel\text{-}aux'\ sf\ xs\ x) * cf\ x)\ xs$
**shows** $asi\ (\lambda l.\ (ldeep\text{-}T\ f\ cf\ l - 1)\ /\ c\ l)\ r\ (c\text{-}list\ f\ cf\ h\ r)$
**unfolding** *asi-def* **using** *c-out-asi-aux-hlist*[*OF assms*] *assms*(*3*) **by** *blast*

**lemma** *asi-if-asi'*: $asi\ rank\ r\ c \Longrightarrow asi'\ r\ c$
**unfolding** *asi'-def asi-def* **by** *auto*

**corollary** *c-out-asi'*:
**assumes** *sel-reasonable sf*
**and** $\forall x.\ cf\ x > 0$
**and** $f = (ldeep\text{-}s\ sf\ xs)$
**and** $h = (\lambda a.\ ldeep\text{-}s\ sf\ xs\ a * cf\ a)$
**shows** $asi'\ r\ (c\text{-}list\ f\ cf\ h\ r)$
**using** *asi-if-asi' c-out-asi*[*OF assms*(*1*,*2*)] *assms*(*3*,*4*) **by** *fast*

**corollary** *c-out-asi'-hlist*:
**assumes** *sel-reasonable sf*
**and** $\forall x.\ cf\ x > 0$
**and** $f = (ldeep\text{-}s\ sf\ xs)$
**and** $h = create\text{-}h\text{-}list\ (\lambda xs\ x.\ (list\text{-}sel\text{-}aux'\ sf\ xs\ x) * cf\ x)\ xs$
**shows** $asi'\ r\ (c\text{-}list\ f\ cf\ h\ r)$
**using** *asi-if-asi' c-out-asi-hlist*[*OF assms*(*1*,*2*)] *assms*(*3*,*4*) **by** *fast*

**lemma** *c-out-asi''-aux*:
**assumes** *sel-reasonable sf*
**and** $\forall x.\ cf\ x > 0$
**and** $c = c\text{-}list\ f\ cf\ h\ r$
**and** $f = (ldeep\text{-}s\ sf\ xs)$
**and** $h = create\text{-}h\text{-}list\ (\lambda xs\ x.\ (list\text{-}sel\text{-}aux'\ sf\ xs\ x) * cf\ x)\ xs$
**and** $distinct\ (A@U@V@B)$
**and** $U \neq []$
**and** $V \neq []$
**and** $rank = (\lambda l.\ (ldeep\text{-}T\ f\ cf\ l - 1)\ /\ c\ l)$
**and** $U \neq [r]$
**and** $V \neq [r]$
**shows** $(c\ (rev\ (A@U@V@B)) \leq c\ (rev\ (A@V@U@B)) \longleftrightarrow rank\ (rev\ U) \leq rank\ (rev\ V))$
**proof** (*cases* $r \notin set\ (A@U@V@B)$)
**case** *True*
**have** *0*: $ldeep\text{-}T\ f\ cf\ (rev\ A) > 0$ **using** *assms*(*1*,*2*,*4*) *ldeep-T-pos* **by** *fast*
**have** $r \notin set\ (rev\ U)$ **using** *True* **by** *simp*
**then have** *1*: $c\text{-}list\ f\ cf\ h\ r\ (rev\ U) > 0$
**using** *c-list-pos-r-not-elem-hlist assms*(*1*,*2*,*4*,*5*,*7*) **by** *fastforce*
**have** $r \notin set\ (rev\ V)$ **using** *True* **by** *simp*
**then have** $c\text{-}list\ f\ cf\ h\ r\ (rev\ V) > 0$
**using** *c-list-pos-r-not-elem-hlist assms*(*1*,*2*,*4*,*5*,*8*) **by** *fastforce*
**then show** *?thesis* **using** *c-list-A-pos-asi 0 1 assms*(*3*,*9*) **by** *fast*
**next**
**case** *False*

**have** *0*: *ldeep-T f cf (rev A) > 0* **using** *assms(1,2,4) ldeep-T-pos* **by** *fast*
**have** *2*: *c-list f cf h r (rev U) > 0*
**using** *c-list-pos-not-root-hlist assms(1,2,4−7,10)* **by** *fastforce*
**have** *c-list f cf h r (rev V) > 0*
**using** *c-list-pos-not-root-hlist assms(1,2,4−6,8,11)* **by** *fastforce*
**then show** *?thesis* **using** *c-list-A-pos-asi 0 2 assms(3,9)* **by** *fast*
**qed**

**theorem** *c-out-asi''*:
**assumes** *sel-reasonable sf*
**and** $\forall x.$ *cf x > 0*
**and** *c = c-list f cf h r*
**and** *f = (ldeep-s sf xs)*
**and** *h = create-h-list (λxs x. (list-sel-aux' sf xs x) ∗ cf x) xs*
**shows** *asi'' (λl. (ldeep-T f cf l − 1) / c l) r (c-list f cf h r)*
**unfolding** *asi''-def* **using** *c-out-asi''-aux*[*OF assms*] *assms(3)* **by** *blast*

### 3.4.2 Additional ASI Proofs

**lemma** *asi-le-iff-notr*:
⟦*asi rank r cost*; *U ≠ []*; *V ≠ []*; *r ∉ set (A @ U @ V @ B)*; *distinct (A @ U @ V @ B)*⟧
⟹ *rank (rev U) ≤ rank (rev V)* ⟷ *cost (rev (A@U@V@B)) ≤ cost (rev (A@V@U@B))*
**unfolding** *asi-def* **by** *blast*

**lemma** *asi-le-iff-rfst*:
⟦*asi rank r cost*; *U ≠ []*; *V ≠ []*;
*take 1 (A @ U @ V @ B) = [r]*; *take 1 (A @ V @ U @ B) = [r]*; *distinct (A @ U @ V @ B)*⟧
⟹ *rank (rev U) ≤ rank (rev V)* ⟷ *cost (rev (A@U@V@B)) ≤ cost (rev (A@V@U@B))*
**unfolding** *asi-def* **by** *blast*

**lemma** *asi-le-notr*:
⟦*asi rank r cost*; *rank (rev U) ≤ rank (rev V)*; *U≠[]*; *V≠[]*;
*distinct (A@U@V@B)*; *r ∉ set (A@U@V@B)*⟧
⟹ *cost (rev (A@U@V@B)) ≤ cost (rev (A@V@U@B))*
**unfolding** *asi-def* **by** *blast*

**lemma** *asi-le-rfst*:
⟦*asi rank r cost*; *rank (rev U) ≤ rank (rev V)*; *U≠[]*; *V≠[]*; *distinct (A@U@V@B)*;
*take 1 (A @ U @ V @ B) = [r]*; *take 1 (A @ V @ U @ B) = [r]*⟧
⟹ *cost (rev (A@U@V@B)) ≤ cost (rev (A@V@U@B))*
**unfolding** *asi-def* **by** *blast*

**lemma** *asi-eq-notr*:
**assumes** *asi rank r cost*
**and** *rank (rev U) = rank (rev V)*

**and** $U \neq []$
**and** $V \neq []$
**and** $r \notin set\ (A@U@V@B)$
**and** $distinct\ (A\ @\ U\ @\ V\ @\ B)$
**shows** $cost\ (rev\ (A@U@V@B)) = cost\ (rev\ (A@V@U@B))$
**proof** −
**have** *0*: $distinct\ (A@V@U@B)$ **using** $assms(6)$ **by** *auto*
**have** *1*: $r \notin set\ (A@V@U@B)$ **using** $assms(5)$ **by** *auto*
**then show** *?thesis*
**using** *asi-le-iff-notr*[*OF assms*(*1*,*3*−*6*)] *asi-le-iff-notr*[*OF assms*(*1*,*4*,*3*) *1 0*] *assms*(*2*) **by** *simp*
**qed**

**lemma** *asi-eq-notr′*:
**assumes** *asi rank r cost*
**and** $cost\ (rev\ (A@U@V@B)) = cost\ (rev\ (A@V@U@B))$
**and** $U \neq []$
**and** $V \neq []$
**and** $r \notin set\ (A@U@V@B)$
**and** $distinct\ (A\ @\ U\ @\ V\ @\ B)$
**shows** $rank\ (rev\ U) = rank\ (rev\ V)$
**proof** −
**have** *0*: $distinct\ (A@V@U@B)$ **using** $assms(6)$ **by** *auto*
**have** *1*: $r \notin set\ (A@V@U@B)$ **using** $assms(5)$ **by** *auto*
**show** *?thesis*
**using** *asi-le-iff-notr*[*OF assms*(*1*,*3*−*6*)] *asi-le-iff-notr*[*OF assms*(*1*,*4*,*3*) *1 0*] *assms*(*2*) **by** *simp*
**qed**

**lemma** *asi-eq-iff-notr*:
$[\![asi\ rank\ r\ cost;\ U \neq []; V \neq []; r \notin set\ (A@U@V@B); distinct\ (A@U@V@B)]\!]$
$\Longrightarrow rank\ (rev\ U) = rank\ (rev\ V) \longleftrightarrow cost\ (rev\ (A@U@V@B)) = cost\ (rev\ (A@V@U@B))$
**using** *asi-eq-notr*[*of rank r cost*] *asi-eq-notr′*[*of rank r cost*] **by** *blast*

**lemma** *asi-eq-rfst*:
**assumes** *asi rank r cost*
**and** $rank\ (rev\ U) = rank\ (rev\ V)$
**and** $U \neq []$
**and** $V \neq []$
**and** $take\ 1\ (A\ @\ U\ @\ V\ @\ B) = [r]$
**and** $take\ 1\ (A\ @\ V\ @\ U\ @\ B) = [r]$
**and** $distinct\ (A\ @\ U\ @\ V\ @\ B)$
**shows** $cost\ (rev\ (A@U@V@B)) = cost\ (rev\ (A@V@U@B))$
**proof** −
**have** *0*: $distinct\ (A@V@U@B)$ **using** $assms(7)$ **by** *auto*
**show** *?thesis*
**using** *asi-le-iff-rfst*[*OF assms*(*1*,*3*−*7*)] *asi-le-iff-rfst*[*OF assms*(*1*,*4*,*3*,*6*,*5*) *0*] *assms*(*2*) **by** *simp*

**qed**

**lemma** *asi-eq-rfst'*:
**assumes** *asi rank r cost*
**and** *cost* (*rev* (*A*@*U*@*V*@*B*)) = *cost* (*rev* (*A*@*V*@*U*@*B*))
**and** *U* ≠ []
**and** *V* ≠ []
**and** *take 1* (*A* @ *U* @ *V* @ *B*) = [*r*]
**and** *take 1* (*A* @ *V* @ *U* @ *B*) = [*r*]
**and** *distinct* (*A* @ *U* @ *V* @ *B*)
**shows** *rank* (*rev U*) = *rank* (*rev V*)
**proof** −
**have** *0*: *distinct* (*A*@*V*@*U*@*B*) **using** *assms*(*7*) **by** *auto*
**show** *?thesis*
**using** *asi-le-iff-rfst*[*OF assms*(*1*,*3*−*7*)] *asi-le-iff-rfst*[*OF assms*(*1*,*4*,*3*,*6*,*5*) *0*] *assms*(*2*) **by** *simp*
**qed**

**lemma** *asi-eq-iff-rfst*:
⟦*asi rank r cost*; *U* ≠ []; *V* ≠ [];
*take 1* (*A* @ *U* @ *V* @ *B*) = [*r*]; *take 1* (*A* @ *V* @ *U* @ *B*) = [*r*]; *distinct* (*A* @ *U* @ *V* @ *B*)⟧
⟹ *rank* (*rev U*) = *rank* (*rev V*) ⟷ *cost* (*rev* (*A*@*U*@*V*@*B*)) = *cost* (*rev* (*A*@*V*@*U*@*B*))
**using** *asi-eq-rfst*[*of rank r cost*] *asi-eq-rfst'*[*of rank r cost*] **by** *blast*

**lemma** *asi-lt-iff-notr*:
**assumes** *asi rank r cost*
**and** *U* ≠ [] **and** *V* ≠ []
**and** *r* ∉ *set* (*A* @ *U* @ *V* @ *B*)
**and** *distinct* (*A* @ *U* @ *V* @ *B*)
**shows** *rank* (*rev U*) < *rank* (*rev V*) ⟷ *cost* (*rev* (*A*@*U*@*V*@*B*)) < *cost* (*rev* (*A*@*V*@*U*@*B*))
**using** *asi-le-iff-notr*[*OF assms*] *asi-eq-iff-notr*[*OF assms*] **by** *auto*

**lemma** *asi-lt-iff-rfst*:
**assumes** *asi rank r cost*
**and** *U* ≠ [] **and** *V* ≠ []
**and** *take 1* (*A* @ *U* @ *V* @ *B*) = [*r*]
**and** *take 1* (*A* @ *V* @ *U* @ *B*) = [*r*]
**and** *distinct* (*A* @ *U* @ *V* @ *B*)
**shows** *rank* (*rev U*) < *rank* (*rev V*) ⟷ *cost* (*rev* (*A*@*U*@*V*@*B*)) < *cost* (*rev* (*A*@*V*@*U*@*B*))
**using** *asi-le-iff-rfst*[*OF assms*] *asi-eq-iff-rfst*[*OF assms*] **by** *auto*

**lemma** *asi-lt-notr*:
⟦*asi rank r cost*; *rank* (*rev U*) < *rank* (*rev V*); *U*≠[]; *V*≠[];
*distinct* (*A*@*U*@*V*@*B*); *r* ∉ *set* (*A*@*U*@*V*@*B*)⟧
⟹ *cost* (*rev* (*A*@*U*@*V*@*B*)) < *cost* (*rev* (*A*@*V*@*U*@*B*))

**using** *asi-lt-iff-notr* **by** *fastforce*

**lemma** *asi-lt-rfst*:
⟦*asi rank r cost*; *rank* (*rev U*) < *rank* (*rev V*); *U*≠[]; *V*≠[]; *distinct* (*A*@*U*@*V*@*B*);
*take 1* (*A* @ *U* @ *V* @ *B*) = [*r*]; *take 1* (*A* @ *V* @ *U* @ *B*) = [*r*]⟧
⟹ *cost* (*rev* (*A*@*U*@*V*@*B*)) < *cost* (*rev* (*A*@*V*@*U*@*B*))
**using** *asi-lt-iff-rfst* **by** *fastforce*

**lemma** *asi''-simp-iff*:
⟦*asi'' rank r cost*; *U* ≠ []; *V* ≠ []; *U* ≠ [*r*]; *V* ≠ [*r*]; *distinct* (*A* @ *U* @ *V* @ *B*)⟧
⟹ *rank* (*rev U*) ≤ *rank* (*rev V*) ⟷ *cost* (*rev* (*A*@*U*@*V*@*B*)) ≤ *cost* (*rev* (*A*@*V*@*U*@*B*))
**unfolding** *asi''-def* **by** *blast*

**lemma** *asi''-simp*:
⟦*asi'' rank r cost*; *rank* (*rev U*) ≤ *rank* (*rev V*); *U*≠[]; *V*≠[]; *distinct* (*A*@*U*@*V*@*B*); *U*≠[*r*]; *V*≠[*r*]⟧
⟹ *cost* (*rev* (*A*@*U*@*V*@*B*)) ≤ *cost* (*rev* (*A*@*V*@*U*@*B*))
**unfolding** *asi''-def* **by** *blast*

**end**

**theory** *Graph-Additions*
**imports** *Complex-Main Graph-Theory.Graph-Theory Shortest-Path-Tree*
**begin**

**lemma** *two-elems-card-ge-2*: *finite xs* ⟹ *x* ∈ *xs* ∧ *y* ∈ *xs* ∧ *x*≠*y* ⟹ *Finite-Set.card xs* ≥ *2*
**using** *card-gt-0-iff mk-disjoint-insert not-less-eq-eq* **by** *fastforce*

# 4 Graph Extensions

**context** *wf-digraph*
**begin**

**lemma** *awalk-dom-if-uneq*: ⟦*u*≠*v*; *awalk u p v*⟧ ⟹ ∃*x*. *x* →$_G$ *v*
**using** *reachable-awalk*[*of u v*] *awalk-ends*[*of u p v*] *converse-reachable-induct* **by** *blast*

**lemma** *awalk-verts-dom-if-uneq*: ⟦*u*≠*v*; *awalk u p v*⟧ ⟹ ∃*x*. *x* →$_G$ *v* ∧ *x* ∈ *set* (*awalk-verts u p*)
**proof**(*induction p arbitrary*: *u*)
**case** *Nil*
**then show** *?case* **using** *awalk-def* **by** *simp*
**next**
**case** (*Cons p ps*)
**then show** *?case*
**using** *awalk-Cons-iff*[*of u p ps v*] *awalk-verts.simps*(*2*)[*of u p ps*] *awalk-verts-non-Nil*

**by** (*metis in-arcs-imp-in-arcs-ends list.sel*(*1*) *list.set-intros*(*2*) *list.set-sel*(*1*))
**qed**

**lemma** *awalk-verts-append-distinct*:
⟦∃ *v*. *awalk r* (*p1*@*p2*) *v*; *distinct* (*awalk-verts r* (*p1*@*p2*))⟧ ⟹ *distinct* (*awalk-verts r p1*)
**using** *awalk-verts-append* **by** *auto*

**lemma** *not-distinct-if-head-eq-tail*:
**assumes** *tail G p* = *u* **and** *head G e* = *u* **and** *awalk r* (*ps*@[*p*]@*e*#*p2*) *v*
**shows** ¬(*distinct* (*awalk-verts r* (*ps*@[*p*]@*e*#*p2*)))
**using** *assms* **proof**(*induction ps arbitrary*: *r*)
**case** *Nil*
**then have** *u* ∈ *set* (*awalk-verts* (*head G p*) (*e*#*p2*))
**by** (*metis append.left-neutral append-Cons awalk-Cons-iff awalk-verts-arc2 list.set-intros*(*1*))
**then show** *?case* **by** (*simp add*: *Nil*(*1*))
**next**
**case** (*Cons p ps*)
**then show** *?case* **using** *awalk-Cons-iff* **by** *auto*
**qed**

**lemma** *awalk-verts-subset-if-p-sub*:
⟦*awalk u p1 v*; *awalk u p2 v*; *set p1* ⊆ *set p2*⟧
⟹ *set* (*awalk-verts u p1*) ⊆ *set* (*awalk-verts u p2*)
**using** *awalk-verts-conv* **by** *fastforce*

**lemma** *awalk-to-apath-verts-subset*:
*awalk u p v* ⟹ *set* (*awalk-verts u* (*awalk-to-apath p*)) ⊆ *set* (*awalk-verts u p*)
**using** *awalk-verts-subset-if-p-sub awalk-to-apath-subset apath-awalk-to-apath awalkI-apath*
**by** *blast*

**lemma** *unique-apath-verts-in-awalk*:
⟦*x* ∈ *set* (*awalk-verts u p1*); *apath u p1 v*; *awalk u p2 v*; ∃!*p*. *apath u p v*⟧
⟹ *x* ∈ *set* (*awalk-verts u p2*)
**using** *apath-awalk-to-apath awalk-to-apath-verts-subset* **by** *blast*

**lemma** *unique-apath-verts-sub-awalk*:
⟦*apath u p v*; *awalk u q v*; ∃!*p*. *apath u p v*⟧ ⟹ *set* (*awalk-verts u p*) ⊆ *set* (*awalk-verts u q*)
**using** *unique-apath-verts-in-awalk* **by** *blast*

**lemma** *awalk-verts-append3*:
⟦*awalk u* (*p*@*e*#*q*) *r*; *awalk v q r*⟧ ⟹ *awalk-verts u* (*p*@*e*#*q*) = *awalk-verts u p* @ *awalk-verts v q*
**using** *awalk-verts-conv* **by** *fastforce*

**lemma** *verts-reachable-connected*:
*verts G* ≠ {} ⟹ (∀ *x*∈*verts G*. ∀ *y*∈*verts G*. *x* →* *y*) ⟹ *connected G*
**by** (*simp add*: *connected-def strongly-connected-def reachable-mk-symmetricI*)

**lemma** *out-degree-0-no-arcs*:
**assumes** *out-degree G v = 0* **and** *finite* (*arcs G*)
**shows** $\forall y.\ (v,y) \notin$ *arcs-ends G*
**proof** (*rule ccontr*)
**assume** $\neg(\forall y.\ (v,y) \notin$ *arcs-ends G*)
**then obtain** *y* **where** *y-def*: $(v,y) \in$ *arcs-ends G* **by** *blast*
**then obtain** *a* **where** *a-def*: $a \in$ *arcs G* $\wedge$ *tail G a = v* $\wedge$ *head G a = y* **by** *auto*
**then have** $a \in \{e \in$ *arcs G*. *tail G e = v*$\}$ **by** *simp*
**then have** *Finite-Set.card* $\{e \in$ *arcs G*. *tail G e = v*$\} > 0$ **using** *assms*(*2*) *card-gt-0-iff* **by** *force*
**then show** *False* **using** *assms*(*1*) **by** (*metis less-nat-zero-code out-arcs-def out-degree-def*)
**qed**

**lemma** *out-degree-0-only-self*: *finite* (*arcs G*) $\Longrightarrow$ *out-degree G v = 0* $\Longrightarrow$ *v* $\rightarrow^*$ *x* $\Longrightarrow$ *x = v*
**using** *converse-reachable-cases out-degree-0-no-arcs* **by** *force*

**lemma** *not-elem-no-out-arcs*: *v* $\notin$ *verts G* $\Longrightarrow$ *out-arcs G v* = $\{\}$
**by** *auto*

**lemma** *not-elem-no-in-arcs*: *v* $\notin$ *verts G* $\Longrightarrow$ *in-arcs G v* = $\{\}$
**by** *auto*

**lemma** *not-elem-out-0*: *v* $\notin$ *verts G* $\Longrightarrow$ *out-degree G v = 0*
**unfolding** *out-degree-def* **using** *not-elem-no-out-arcs* **by** *simp*

**lemma** *not-elem-in-0*: *v* $\notin$ *verts G* $\Longrightarrow$ *in-degree G v = 0*
**unfolding** *in-degree-def* **using** *not-elem-no-in-arcs* **by** *simp*

**lemma** *new-vert-only-no-arcs*:
**assumes** *G* = $(\!|$*verts = V* $\cup$ $\{v\}$, *arcs = A*, *tail = t*, *head = h*$|\!)$
**and** *G′* = $(\!|$*verts = V*, *arcs = A*, *tail = t*, *head = h*$|\!)$
**and** *wf-digraph G′*
**and** *v* $\notin$ *V*
**and** *finite* (*arcs G*)
**shows** $\forall u.\ (v,u) \notin$ *arcs-ends G*
**proof** −
**have** *out-degree G′ v = 0* **using** *assms*(*2−4*) *wf-digraph.not-elem-out-0* **by** *fastforce*
**then have** *out-degree G v = 0* **unfolding** *out-degree-def out-arcs-def* **using** *assms*(*1*,*2*) **by** *simp*
**then show** *?thesis* **using** *assms*(*5*) *out-degree-0-no-arcs* **by** *blast*
**qed**

**lemma** *new-leaf-out-sets-eq*:
**assumes** *G* = $(\!|$*verts = V* $\cup$ $\{v\}$, *arcs = A* $\cup$ $\{a\}$, *tail = t*(*a* := *u*), *head = h*(*a* := *v*)$|\!)$
**and** *G′* = $(\!|$*verts = V*, *arcs = A*, *tail = t*, *head = h*$|\!)$

```
    and u ∈ V
    and v ∉ V
    and a ∉ A
  shows {e ∈ arcs G. tail G e = v} = {e ∈ arcs G'. tail G' e = v}
 using assms by auto

lemma new-leaf-out-0:
 assumes G = (|verts = V ∪ {v}, arcs = A ∪ {a}, tail = t(a := u), head = h(a
:= v)|)
    and G' = (|verts = V, arcs = A, tail = t, head = h|)
    and wf-digraph G'
    and u ∈ V
    and v ∉ V
    and a ∉ A
  shows out-degree G v = 0
proof −
 have tail G a = u using assms(1) by simp
 then have 0: {e ∈ arcs G. tail G e = v} = {e ∈ arcs G'. tail G' e = v}
  using new-leaf-out-sets-eq assms(1,2,4−6) by blast
 have out-degree G' v = 0 using assms(2,3,5) wf-digraph.not-elem-out-0 by
fastforce
 then show ?thesis unfolding out-degree-def out-arcs-def using 0 by simp
qed

lemma new-leaf-no-arcs:
 assumes G = (|verts = V ∪ {v}, arcs = A ∪ {a}, tail = t(a := u), head = h(a
:= v)|)
    and G' = (|verts = V, arcs = A, tail = t, head = h|)
    and wf-digraph G'
    and u ∈ V
    and v ∉ V
    and a ∉ A
    and finite (arcs G)
  shows ∀ u. (v,u) ∉ arcs-ends G
 using new-leaf-out-0 assms out-degree-0-no-arcs by presburger

lemma tail-and-head-eq-impl-cas:
 assumes cas x p y
    and ∀ x ∈ set p. tail G x = tail G' x
    and ∀ x ∈ set p. head G x = head G' x
  shows pre-digraph.cas G' x p y
using assms proof(induction p arbitrary: x y)
 case Nil
 show ?case using pre-digraph.cas.simps(1) Nil(1) by fastforce
next
 case (Cons p ps)
 have 0: tail G' p = x using Cons.prems(1,2) by simp
 have cas (head G p) ps y using Cons.prems(1) by simp
 then have pre-digraph.cas G' (head G' p) ps y using Cons.IH Cons.prems(2,3)
```

**by** *simp*
**then show** *?case* **using** *0* **by** (*simp add*: *pre-digraph.cas.simps*(*2*))
**qed**

**lemma** *new-leaf-same-reachables-orig*:
**assumes** $x \rightarrow^*_G y$
**and** $G = (\!|verts = V \cup \{v\}, arcs = A \cup \{a\}, tail = t(a := u), head = h(a := v)|\!)$
**and** $G' = (\!|verts = V, arcs = A, tail = t, head = h|\!)$
**and** *wf-digraph* $G'$
**and** $x \in V$
**and** $u \in V$
**and** $v \notin V$
**and** $y \neq v$
**and** $a \notin A$
**and** *finite* (*arcs* $G$)
**shows** $x \rightarrow^*_{G'} y$
**proof** −
**obtain** $p$ **where** *p-def*: *awalk* $x$ $p$ $y$ **using** *reachable-awalk assms*(*1*) **by** *auto*
**then have** *0*: *set* $p \subseteq$ *arcs* $G$ **by** *blast*
**have** *v-0*: *out-degree* $G$ $v = 0$ **using** *new-leaf-out-0 assms* **by** *presburger*
**have** *a-notin-p*: $a \notin$ *set* $p$
**proof**
**assume** *asm*: $a \in$ *set* $p$
**have** *head* $G$ $a = v$ **using** *assms*(*2*) **by** *simp*
**then have** $\exists p'\ p''.\ p'@p''=p \wedge$ *awalk* $x$ $p'$ $v$
**using** *asm awalk-decomp awalk-verts-arc2 p-def* **by** *metis*
**then obtain** $p'$ $p''$ **where** *p'-def*: $p'@p''=p \wedge$ *awalk* $x$ $p'$ $v$ **by** *blast*
**then have** *awalk* $v$ $p''$ $y$ **using** *p-def* **by** *auto*
**then have** $v \rightarrow^* y$ **using** *reachable-awalk* **by** *auto*
**then have** $v = y$ **using** *out-degree-0-only-self assms*(*10*) *v-0* **by** *blast*
**then show** *False* **using** *assms*(*8*) **by** *simp*
**qed**
**then have** *1*: *set* $p \subseteq$ *arcs* $G'$ **using** *assms*(*2*,*3*) *0* **by** *auto*
**have** $\forall x \in$ *set* $p$. *tail* $G$ $x$ = *tail* $G'$ $x$ **using** *assms*(*2*,*3*) *a-notin-p* **by** *simp*
**moreover have** $\forall x \in$ *set* $p$. *head* $G$ $x$ = *head* $G'$ $x$ **using** *assms*(*2*,*3*) *a-notin-p* **by** *simp*
**ultimately have** *pre-digraph.cas* $G'$ $x$ $p$ $y$ **using** *tail-and-head-eq-impl-cas p-def* **by** *blast*
**then have** *pre-digraph.awalk* $G'$ $x$ $p$ $y$ **unfolding** *pre-digraph.awalk-def* **using** *assms*(*3*,*5*) *1* **by** *simp*
**then show** *?thesis* **using** *assms*(*4*) *wf-digraph.reachable-awalkI* **by** *fast*
**qed**

**lemma** *new-leaf-same-reachables-new*:
**assumes** $x \rightarrow^*_{G'} y$
**and** $G = (\!|verts = V \cup \{v\}, arcs = A \cup \{a\}, tail = t(a := u), head = h(a := v)|\!)$
**and** $G' = (\!|verts = V, arcs = A, tail = t, head = h|\!)$

**and** *wf-digraph* $G'$
**and** $x \in V$
**and** $u \in V$
**and** $v \notin V$
**and** $y \neq v$
**and** $a \notin A$
**shows** $x \rightarrow^*_G y$
**proof** −
**obtain** $p$ **where** *p-def*: *pre-digraph.awalk* $G'$ $x$ $p$ $y$
**using** *wf-digraph.reachable-awalk assms*(*1*,*4*) **by** *fast*
**then have** *0*: *set* $p \subseteq$ *arcs* $G'$ **by** (*meson pre-digraph.awalk-def*)
**then have** *a-notin-p*: $a \notin$ *set* $p$ **using** *assms*(*3*,*9*) **by** *auto*
**have** *1*: *set* $p \subseteq$ *arcs* $G$ **using** *assms*(*2*,*3*) *0* **by** *auto*
**have** $\forall x \in$ *set* $p$. *tail* $G$ $x$ = *tail* $G'$ $x$ **using** *assms*(*2*,*3*) *a-notin-p* **by** *simp*
**moreover have** $\forall x \in$ *set* $p$. *head* $G$ $x$ = *head* $G'$ $x$ **using** *assms*(*2*,*3*) *a-notin-p*
**by** *simp*
**moreover have** *pre-digraph.cas* $G'$ $x$ $p$ $y$ **using** *p-def pre-digraph.awalk-def* **by**
*fast*
**ultimately have** *cas* $x$ $p$ $y$ **using** *assms*(*4*) *wf-digraph.tail-and-head-eq-impl-cas*
**by** *fastforce*
**then have** *awalk* $x$ $p$ $y$ **unfolding** *awalk-def* **using** *assms*(*2*,*5*) *1* **by** *simp*
**then show** *?thesis* **using** *reachable-awalkI* **by** *simp*
**qed**

**lemma** *new-leaf-reach-impl-parent*:
**assumes** $y \rightarrow^* v$
**and** $G = (\!|verts = V \cup \{v\},\ arcs = A \cup \{a\},\ tail = t(a := u),\ head = h(a$
$:= v)|\!)$
**and** $G' = (\!|verts = V,\ arcs = A,\ tail = t,\ head = h|\!)$
**and** *wf-digraph* $G'$
**and** $y \in V$
**and** $v \notin V$
**shows** $y \rightarrow^* u$
**proof** −
**have** $\forall a \in A$. $h\ a \neq v$
**using** *assms*(*3*,*4*,*6*) *wf-digraph.head-in-verts* **by** (*metis pre-digraph.select-convs*(*1*,*2*,*4*))
**then have** *0*: $\forall x$. $(x,v) \in$ *arcs-ends* $G \longrightarrow x = u$ **using** *assms*(*2*) **by** *fastforce*
**have** $v \neq y$ **using** *assms*(*5*,*6*) **by** *blast*
**then have** $y \rightarrow^+ v$ **using** *assms*(*1*) **by** *blast*
**then have** $\exists x$. $y \rightarrow^* x \wedge x \rightarrow_G v$
**by** (*meson reachable1-in-verts*(*1*) *reachable-conv'* *tranclD2*)
**then obtain** $x$ **where** $y \rightarrow^* x \wedge x \rightarrow_G v$ **by** *blast*
**then show** *?thesis* **using** *0* **by** *blast*
**qed**

**end**

**context** *graph*
**begin**

**abbreviation** *min-degree* :: *'a set* ⇒ *'a* ⇒ *bool* **where**
*min-degree xs x* ≡ *x*∈*xs* ∧ (∀ *y*∈*xs*. *out-degree G x* ≤ *out-degree G y*)

**lemma** *graph-del-vert-sym*: *sym* (*arcs-ends* (*del-vert x*))
**by** (*smt* (*z3*) *wf-digraph-del-vert mem-Collect-eq reachableE sym-digraph-axioms-def arcs-del-vert*
*symmetric-conv symI wf-digraph.in-arcs-imp-in-arcs-ends head-del-vert sym-arcs tail-del-vert*)

**lemma** *graph-del-vert*: *graph* (*del-vert x*)
**apply**(*standard*)
**by** (*auto simp*: *arcs-del-vert2 tail-del-vert head-del-vert verts-del-vert*
*no-loops ends-del-vert no-multi-arcs symmetric-def graph-del-vert-sym*)

**lemma** *connected-iff-reachable*:
*connected G* ⟷ ((∀ *x*∈*verts G*. ∀ *y*∈*verts G*. *x* →* *y*) ∧ *verts G* ≠ {})
**using** *symmetric-connected-imp-strongly-connected strongly-connected-def verts-reachable-connected*
**by**(*blast*)

**end**

**context** *nomulti-digraph*
**begin**

**lemma** *no-multi-alt*:
⟦*e1* ∈ *arcs G*; *e2* ∈ *arcs G*; *e1* ≠ *e2*⟧ ⟹ *head G e1* ≠ *head G e2* ∨ *tail G e1* ≠ *tail G e2*
**using** *no-multi-arcs* **by**(*auto simp*: *arc-to-ends-def*)

**end**

## 4.1 Vertices with Multiple Outgoing Arcs

**context** *wf-digraph*
**begin**

**definition** *branching-points* :: *'a set* **where**
*branching-points* = {*x*. ∃ *y*∈*arcs G*. ∃ *z*∈*arcs G*. *y*≠*z* ∧ *tail G y* = *x* ∧ *tail G z* = *x*}

**definition** *is-chain* :: *bool* **where**
*is-chain* = (*branching-points* = {})

**definition** *last-branching-points* :: *'a set* **where**
*last-branching-points* = {*x*. (*x*∈*branching-points* ∧ ¬(∃ *y* ∈ *branching-points*. *y*≠*x* ∧ *x* →* *y*))}

**lemma** *branch-in-verts*: *x* ∈ *branching-points* ⟹ *x* ∈ *verts G*

**unfolding** *branching-points-def* **by** *auto*

**lemma** *last-branch-is-branch*:
(*y*∈*last-branching-points* ⟹ *y*∈*branching-points*)
**unfolding** *last-branching-points-def* **by** *blast*

**lemma** *last-branch-alt*: *x* ∈ *last-branching-points* ⟹ (∀ *z*. *x* →* *z* ∧ *z*≠*x* ⟶ *z* ∉ *branching-points*)
**unfolding** *last-branching-points-def* **by** *blast*

**lemma** *braching-points-alt*:
**assumes** *finite* (*arcs G*)
**shows** *x* ∈ *branching-points* ⟷ *out-degree G x* ≥ *2* (**is** *?P* ⟷ *?Q*)
**proof**
**assume** *?P*
**then obtain** *a1 a2* **where** *a1*∈*arcs G* ∧ *a2*∈*arcs G* ∧ *a1*≠*a2* ∧ *tail G a1* = *x* ∧ *tail G a2* = *x*
**using** *branching-points-def* **by** *auto*
**then have** *0*: *a1* ∈ *out-arcs G x* ∧ *a2* ∈ *out-arcs G x* ∧ *a1*≠*a2* **by** *simp*
**have** *finite* (*out-arcs G x*) **by** (*simp add*: *assms out-arcs-def*)
**then show** *?Q* **unfolding** *out-degree-def* **using** *0 two-elems-card-ge-2* **by** *fast*
**next**
**assume** *0*: *?Q*
**have** *finite* (*out-arcs G x*) **by** (*simp add*: *assms out-arcs-def*)
**then have** ∃ *a1 a2*. *a1* ∈ (*out-arcs G x*) ∧ *a2* ∈ (*out-arcs G x*) ∧ *a1*≠*a2*
**using** *0 out-degree-def* **by** (*metis Suc-n-not-le-n card-le-Suc0-iff-eq le-trans numeral-2-eq-2*)
**then show** *?P* **unfolding** *branching-points-def* **by** *auto*
**qed**

**lemma** *branch-in-supergraph*:
**assumes** *subgraph C G*
**and** *x* ∈ *wf-digraph.branching-points C*
**shows** *x* ∈ *branching-points*
**proof** −
**have** *0*: *wf-digraph C* **using** *assms*(*1*) *Digraph-Component.subgraph-def subgraph.sub-G* **by** *auto*
**have** *1*: *wf-digraph G* **using** *assms*(*1*) *subgraph.sub-G* **by** *auto*
**obtain** *y z* **where** *arcs-C*: *y*∈*arcs C* ∧ *z*∈*arcs C* ∧ *y*≠*z* ∧ *tail C y* = *x* ∧ *tail C z* = *x*
**using** *assms*(*2*) *wf-digraph.branching-points-def 0* **by** *blast*
**then have** *y*∈*arcs G* ∧ *z*∈*arcs G* ∧ *y*≠*z* ∧ *tail C y* = *x* ∧ *tail C z* = *x*
**using** *assms*(*1*) *subgraph.sub-G* **by** *blast*
**then have** *y*∈*arcs G* ∧ *z*∈*arcs G* ∧ *y*≠*z* ∧ *tail G y* = *x* ∧ *tail G z* = *x*
**using** *assms*(*1*) *subgraph.sub-G compatible-def* **by** *force*
**then show** *?thesis* **using** *branching-points-def assms*(*1*) *subgraph.sub-G* **by** *blast*
**qed**

**lemma** *subgraph-no-branch-chain*:

**assumes** *subgraph C G*
**and** *verts C* $\subseteq$ *verts G* $-$ $\{x.\ \exists\, y{\in}$*branching-points*$.\ x \rightarrow^*_G y\}$
**shows** *wf-digraph.is-chain C*
**proof** (*rule ccontr*)
**assume** *asm*: $\neg$*wf-digraph.is-chain C*
**let** *?rem* = $\{x.\ \exists\, y{\in}$*branching-points*$.\ x \rightarrow^*_G y\}$
**have** *wf-digraph C* **using** *assms*(*1*) *Digraph-Component.subgraph-def subgraph.sub-G*
**by** *auto*
**then obtain** *x* **where** *x-def*[*simp*]: *x* $\in$ *wf-digraph.branching-points C*
**using** *wf-digraph.is-chain-def asm* **by** *blast*
**then have** *x* $\in$ *branching-points* **using** *assms*(*1*) *branch-in-supergraph* **by** *simp*
**moreover from** *this* **have** *x* $\in$ *verts G* **using** *branch-in-verts* **by** *simp*
**moreover from** *this* **have** *x* $\rightarrow^*_G$ *x* **by** *simp*
**ultimately have** *x* $\in$ *?rem* **by** *blast*
**then show** *False* **using** *assms*(*2*) ‹*wf-digraph C*› *subsetD wf-digraph.branch-in-verts*
**by** *fastforce*
**qed**

**lemma** *branch-if-leaf-added*:
**assumes** *x*$\in$*wf-digraph.branching-points G'*
**and** *G* = (|*verts* = *V* $\cup$ {*v*}, *arcs* = *A* $\cup$ {*a*}, *tail* = *t*(*a* := *u*), *head* = *h*(*a* := *v*)|)
**and** *G'* = (|*verts* = *V*, *arcs* = *A*, *tail* = *t*, *head* = *h*|)
**and** *wf-digraph G'*
**and** *a* $\notin$ *A*
**shows** *x* $\in$ *branching-points*
**proof** $-$
**obtain** *a1 a2* **where** *a12*: *a1*$\in$*arcs G'* $\wedge$ *a2*$\in$*arcs G'* $\wedge$ *a1*$\neq$*a2* $\wedge$ *tail G' a1* = *x* $\wedge$ *tail G' a2* = *x*
**using** *wf-digraph.branching-points-def assms*(*1*,*4*) **by** *blast*
**then have** *a1* $\neq$ *a* $\wedge$ *a2* $\neq$ *a* **using** *assms*(*3*,*5*) **by** *auto*
**then have** *0*: *tail G a1* = *tail G' a1* $\wedge$ *tail G a2* = *tail G' a2* **using** *assms*(*2*,*3*)
**by** *simp*
**have** *a1*$\in$*arcs G* $\wedge$ *a2*$\in$*arcs G* $\wedge$ *a1*$\neq$*a2* $\wedge$ *a1*$\neq$*a2* $\wedge$ *tail G' a1* = *x* $\wedge$ *tail G' a2* = *x*
**using** *assms*(*2*,*3*) *a12* **by** *simp*
**then have** *a1*$\in$*arcs G* $\wedge$ *a2*$\in$*arcs G* $\wedge$ *a1*$\neq$*a2* $\wedge$ *tail G a1* = *x* $\wedge$ *tail G a2* = *x*
**using** *0* **by** *simp*
**then show** *?thesis* **unfolding** *branching-points-def* **by** *blast*
**qed**

**lemma** *new-leaf-no-branch*:
**assumes** *G* = (|*verts* = *V* $\cup$ {*v*}, *arcs* = *A* $\cup$ {*a*}, *tail* = *t*(*a* := *u*), *head* = *h*(*a* := *v*)|)
**and** *G'* = (|*verts* = *V*, *arcs* = *A*, *tail* = *t*, *head* = *h*|)
**and** *wf-digraph G'*
**and** *u* $\in$ *V*
**and** *v* $\notin$ *V*
**and** *a* $\notin$ *A*

**shows** $v \notin$ *branching-points*
**proof** −
**have** $v \neq u$ **using** *assms*(4,5) **by** *fast*
**have** $\forall\, a{\in}arcs\ G'.\ tail\ G'\ a \neq v$
**using** *assms*(2,3,5) *pre-digraph.select-convs*(1) *wf-digraph-def* **by** *fast*
**moreover have** $\forall\, x \in arcs\ G'.\ tail\ G\ x = tail\ G'\ x$ **using** *assms*(1,2,6) **by** *simp*
**ultimately have** $\forall\, a{\in}arcs\ G'.\ tail\ G\ a \neq v$ **by** *simp*
**then have** $\forall\, a{\in}arcs\ G.\ tail\ G\ a \neq v$
**using** *assms*(1,2,6) *Un-iff pre-digraph.select-convs*(2) *singletonD* ‹$v \neq u$› **by** *simp*
**then show** *?thesis* **unfolding** *branching-points-def* **by** *blast*
**qed**

**lemma** *new-leaf-not-reach-last-branch*:
**assumes** $y{\in}wf\text{-}digraph.last\text{-}branching\text{-}points\ G'$
**and** $\neg\ y \rightarrow^* u$
**and** $G = (\!|verts = V \cup \{v\},\ arcs = A \cup \{a\},\ tail = t(a := u),\ head = h(a := v)|\!)$
**and** $G' = (\!|verts = V,\ arcs = A,\ tail = t,\ head = h|\!)$
**and** *wf-digraph* $G'$
**and** $y \in V$
**and** $u \in V$
**and** $v \notin V$
**and** $a \notin A$
**and** *finite* (*arcs* $G$)
**shows** $\neg(\exists\, z \in branching\text{-}points.\ z{\neq}y \wedge y \rightarrow^* z)$
**proof**
**assume** $\exists\, z \in branching\text{-}points.\ z{\neq}y \wedge y \rightarrow^* z$
**then obtain** $z$ **where** *z-def*: $z \in branching\text{-}points \wedge z{\neq}y \wedge y \rightarrow^* z$ **by** *blast*
**then have** $z \neq u$ **using** *assms*(2) **by** *blast*
**then obtain** *a1 a2* **where** *a12*: $a1{\in}arcs\ G \wedge a2{\in}arcs\ G \wedge a1{\neq}a2 \wedge tail\ G\ a1 = z \wedge tail\ G\ a2 = z$
**using** *branching-points-def z-def* **by** *blast*
**then have** *0*: $a1 \neq a \wedge a2 \neq a$ **using** *assms*(3) ‹$z{\neq}u$› **by** *fastforce*
**then have** *1*: $tail\ G\ a1 = tail\ G'\ a1 \wedge tail\ G\ a2 = tail\ G'\ a2$ **using** *assms*(3,4) **by** *simp*
**have** $a1{\in}arcs\ G' \wedge a2{\in}arcs\ G' \wedge a1{\neq}a2 \wedge tail\ G\ a1 = z \wedge tail\ G\ a2 = z$
**using** *assms*(3,4) *a12 0* **by** *simp*
**then have** $a1{\in}arcs\ G' \wedge a2{\in}arcs\ G' \wedge a1{\neq}a2 \wedge tail\ G'\ a1 = z \wedge tail\ G'\ a2 = z$
**using** *1* **by** *simp*
**then have** *2*: $z \in wf\text{-}digraph.branching\text{-}points\ G'$
**using** *wf-digraph.branching-points-def assms*(5) **by** *auto*
**have** $z \neq v$ **using** *assms*(2,3,4,5,6,8) *z-def new-leaf-reach-impl-parent* **by** *blast*
**then have** $y \rightarrow^*_{G'} z$ **using** *new-leaf-same-reachables-orig z-def assms* **by** *blast*
**then have** $\exists\, z{\in}wf\text{-}digraph.branching\text{-}points\ G'.\ z{\neq}y \wedge y \rightarrow^*_{G'} z$ **using** *2 z-def* **by** *blast*
**then have** $y \notin wf\text{-}digraph.last\text{-}branching\text{-}points\ G'$
**using** *wf-digraph.last-branching-points-def assms*(5) **by** *blast*

**then show** *False* **using** *assms(1)* **by** *simp*
**qed**

**lemma** *new-leaf-parent-nbranch-in-orig*:
**assumes** $y \in$ *branching-points*
**and** $y \neq u$
**and** $G = (\!|verts = V \cup \{v\},\ arcs = A \cup \{a\},\ tail = t(a := u),\ head = h(a := v)|\!)$
**and** $G' = (\!|verts = V,\ arcs = A,\ tail = t,\ head = h|\!)$
**and** *wf-digraph* $G'$
**shows** $y \in$ *wf-digraph.branching-points* $G'$
**proof** −
**obtain** *a1 a2* **where** *a12*: $a1 \in arcs\ G \wedge a2 \in arcs\ G \wedge a1 \neq a2 \wedge tail\ G\ a1 = y \wedge tail\ G\ a2 = y$
**using** *branching-points-def assms(1)* **by** *blast*
**then have** *0*: $a1 \neq a \wedge a2 \neq a$ **using** *assms(2,3)* **by** *fastforce*
**then have** *1*: $tail\ G\ a1 = tail\ G'\ a1 \wedge tail\ G\ a2 = tail\ G'\ a2$ **using** *assms(3,4)* **by** *simp*
**have** $a1 \in arcs\ G' \wedge a2 \in arcs\ G' \wedge a1 \neq a2 \wedge tail\ G\ a1 = y \wedge tail\ G\ a2 = y$
**using** *assms(3,4) a12 0* **by** *auto*
**then have** $a1 \in arcs\ G' \wedge a2 \in arcs\ G' \wedge a1 \neq a2 \wedge tail\ G'\ a1 = y \wedge tail\ G'\ a2 = y$
**using** *1* **by** *simp*
**then show** *?thesis* **using** *assms(5) wf-digraph.branching-points-def* **by** *auto*
**qed**

**lemma** *new-leaf-last-branch-exists-preserv*:
**assumes** $y \in$ *wf-digraph.last-branching-points* $G'$
**and** $x \rightarrow^* y$
**and** $G = (\!|verts = V \cup \{v\},\ arcs = A \cup \{a\},\ tail = t(a := u),\ head = h(a := v)|\!)$
**and** $G' = (\!|verts = V,\ arcs = A,\ tail = t,\ head = h|\!)$
**and** *wf-digraph* $G'$
**and** $y \in V$
**and** $u \in V$
**and** $v \notin V$
**and** $a \notin A$
**and** *finite* $(arcs\ G)$
**and** $\forall x.\ y \rightarrow^+ x \longrightarrow y \neq x$
**obtains** $y'$ **where** $y' \in$ *last-branching-points* $\wedge\ x \rightarrow^* y'$
**proof** (*cases* $y \rightarrow^* u$)
**case** *True*
**have** $y \in$ *wf-digraph.branching-points* $G'$
**using** *assms(1,5) wf-digraph.last-branch-is-branch* **by** *fast*
**then have** *y-branch*: $y \in$ *branching-points* **using** *branch-if-leaf-added assms(3−5,9)* **by** *blast*
**have** *v-nbranch*: $v \notin$ *branching-points* **using** *new-leaf-no-branch assms(3−5,7−9)* **by** *blast*
**then show** *?thesis*

**proof**(*cases* $u \in$ *branching-points*)
**case** *True*
**have** $\neg(\exists\, z \in$ *branching-points*. $z{\neq}u \wedge u \rightarrow^* z)$
**proof**
**assume** $\exists\, z \in$ *branching-points*. $z{\neq}u \wedge u \rightarrow^* z$
**then obtain** $z$ **where** *z-def*: $z \in$ *branching-points* $\wedge$ $z{\neq}u \wedge u \rightarrow^* z$ **by** *blast*
**then have** $z \neq v$ **using** *v-nbranch* **by** *blast*
**then have** $u \rightarrow^*{}_{G'} z$
**using** *new-leaf-same-reachables-orig assms*(*3*−*5*,*7*−*10*) *z-def* **by** *blast*
**moreover have** $y \rightarrow^*{}_{G'} u$
**using** *new-leaf-same-reachables-orig* ‹$y \rightarrow^* u$› *assms*(*3*−*10*) **by** *blast*
**ultimately have** *0*: $y \rightarrow^*{}_{G'} z$
**using** *assms*(*5*) *wf-digraph.reachable-trans* **by** *fast*
**have** $y \rightarrow^+ z$
**using** ‹$y \rightarrow^* u$› *z-def reachable-reachable1-trans reachable-neq-reachable1* **by** *blast*
**then have** $y \neq z$ **using** *assms*(*11*) **by** *simp*
**have** $z \in$ *wf-digraph.branching-points* $G'$
**using** *z-def new-leaf-parent-nbranch-in-orig assms*(*3*−*5*) **by** *blast*
**then have** $y \notin$ *wf-digraph.last-branching-points* $G'$
**using** *0 assms*(*5*) *wf-digraph.last-branch-alt* ‹$y \neq z$› **by** *fast*
**then show** *False* **using** *assms*(*1*) **by** *simp*
**qed**
**then have** $u \in$ *last-branching-points* **unfolding** *last-branching-points-def* **using** *True* **by** *blast*
**then show** *?thesis* **using** *assms*(*2*) ‹$y \rightarrow^* u$› *reachable-trans that* **by** *blast*
**next**
**case** *False*
**have** $\neg(\exists\, z \in$ *branching-points*. $z{\neq}y \wedge y \rightarrow^* z)$
**proof**
**assume** $\exists\, z \in$ *branching-points*. $z{\neq}y \wedge y \rightarrow^* z$
**then obtain** $z$ **where** *z-def*: $z \in$ *branching-points* $\wedge$ $z{\neq}y \wedge y \rightarrow^* z$ **by** *blast*
**then have** $z \neq v$ **using** *v-nbranch* **by** *blast*
**then have** *0*: $y \rightarrow^*{}_{G'} z$
**using** *new-leaf-same-reachables-orig assms*(*3*−*10*) *z-def* **by** *blast*
**have** $z \neq u$ **using** *False z-def* **by** *blast*
**then have** $z \in$ *wf-digraph.branching-points* $G'$
**using** *z-def new-leaf-parent-nbranch-in-orig assms*(*3*−*5*) **by** *blast*
**then have** $y \notin$ *wf-digraph.last-branching-points* $G'$
**using** *0 z-def assms*(*5*) *wf-digraph.last-branch-alt* **by** *fast*
**then show** *False* **using** *assms*(*1*) **by** *simp*
**qed**
**then have** $y \in$ *last-branching-points* **using** *last-branching-points-def y-branch* **by** *simp*
**then show** *?thesis* **using** *assms*(*2*) *that* **by** *blast*
**qed**
**next**
**case** *False*
**have** $y \in$ *wf-digraph.branching-points* $G'$

**using** *assms*(*1*,*5*) *wf-digraph.last-branch-is-branch* **by** *fast*
**then have** $y \in$ *branching-points* **using** *branch-if-leaf-added assms*(*3*−*5*,*9*) **by** *blast*
**moreover have** $\neg(\exists\, z \in$ *branching-points*. $z{\neq}y \wedge y \rightarrow^* z)$
**using** *new-leaf-not-reach-last-branch assms*(*1*,*3*−*10*) *False* **by** *blast*
**ultimately have** $y \in$ *last-branching-points* **unfolding** *last-branching-points-def* **by** *blast*
**then show** *?thesis* **using** *assms*(*2*) *that* **by** *blast*
**qed**

**end**

## 4.2 Vertices with Multiple Incoming Arcs

**context** *wf-digraph*
**begin**

**definition** *merging-points* :: *'a set* **where**
*merging-points* = $\{x.\ \exists\, y{\in}$*arcs G*. $\exists\, z{\in}$*arcs G*. $y{\neq}z \wedge$ *head G y* = *x* $\wedge$ *head G z* = *x*$\}$

**definition** *is-chain'* :: *bool* **where**
*is-chain'* = (*merging-points* = {})

**definition** *last-merging-points* :: *'a set* **where**
*last-merging-points* = $\{x.\ (x{\in}$*merging-points* $\wedge\ \neg(\exists\, y \in$ *merging-points*. $y{\neq}x \wedge x \rightarrow^* y))\}$

**lemma** *merge-in-verts*: $x \in$ *merging-points* $\Longrightarrow x \in$ *verts G*
**unfolding** *merging-points-def* **by** *auto*

**lemma** *last-merge-is-merge*:
($y{\in}$*last-merging-points* $\Longrightarrow$ $y{\in}$*merging-points*)
**unfolding** *last-merging-points-def* **by** *blast*

**lemma** *last-merge-alt*: $x \in$ *last-merging-points* $\Longrightarrow (\forall\, z.\ x \rightarrow^* z \wedge z{\neq}x \longrightarrow z \notin$ *merging-points*)
**unfolding** *last-merging-points-def* **using** *reachable-in-verts*(*2*) **by** *blast*

**lemma** *merge-in-supergraph*:
**assumes** *subgraph C G*
**and** $x \in$ *wf-digraph.merging-points C*
**shows** $x \in$ *merging-points*
**proof** −
**have** *0*: *wf-digraph C* **using** *assms*(*1*) *Digraph-Component.subgraph-def subgraph.sub-G* **by** *auto*
**have** *1*: *wf-digraph G* **using** *assms*(*1*) *subgraph.sub-G* **by** *auto*
**obtain** *y z* **where** *arcs-C*: $y{\in}$*arcs C* $\wedge$ $z{\in}$*arcs C* $\wedge$ $y{\neq}z \wedge$ *head C y* = *x* $\wedge$ *head C z* = *x*

**using** *assms(2) wf-digraph.merging-points-def 0* **by** *blast*
**then have** *y∈arcs G ∧ z∈arcs G ∧ y≠z ∧ head C y = x ∧ head C z = x*
**using** *assms(1) subgraph.sub-G* **by** *blast*
**then have** *y∈arcs G ∧ z∈arcs G ∧ y≠z ∧ head G y = x ∧ head G z = x*
**using** *assms(1) subgraph.sub-G compatible-def* **by** *force*
**then show** *?thesis* **using** *merging-points-def assms(1) subgraph.sub-G* **by** *blast*
**qed**

**lemma** *subgraph-no-merge-chain*:
**assumes** *subgraph C G*
**and** *verts C ⊆ verts G − {x. ∃ y∈merging-points. x →*$^*_G$ *y}*
**shows** *wf-digraph.is-chain′ C*
**proof** (*rule ccontr*)
**assume** *asm*: *¬wf-digraph.is-chain′ C*
**let** *?rem = {x. ∃ y∈merging-points. x →*$^*_G$ *y}*
**have** *wf-digraph C* **using** *assms(1) Digraph-Component.subgraph-def subgraph.sub-G*
**by** *auto*
**then obtain** *x* **where** *x-def*[*simp*]: *x ∈ wf-digraph.merging-points C*
**using** *wf-digraph.is-chain′-def asm* **by** *blast*
**then have** *x ∈ merging-points* **using** *assms(1) merge-in-supergraph* **by** *simp*
**moreover from** *this* **have** *x ∈ verts G* **using** *merge-in-verts* **by** *simp*
**moreover from** *this* **have** *x →*$^*_G$ *x* **by** *simp*
**ultimately have** *x ∈ ?rem* **by** *blast*
**then show** *False* **using** *assms(2) ‹wf-digraph C› subsetD wf-digraph.merge-in-verts*
**by** *fastforce*
**qed**

**end**

**end**

**theory** *QueryGraph*
**imports** *Complex-Main Graph-Additions Selectivities JoinTree*
**begin**

# 5 Query Graphs

**locale** *query-graph = graph +*
**fixes** *sel :: 'b weight-fun*
**fixes** *cf :: 'a ⇒ real*
**assumes** *sel-sym*: *⟦tail G e*$_1$ *= head G e*$_2$*; head G e*$_1$ *= tail G e*$_2$*⟧ ⟹ sel e*$_1$ *= sel e*$_2$
**and** *not-arc-sel-1*: *e ∉ arcs G ⟹ sel e = 1*
**and** *sel-pos*: *sel e > 0*
**and** *sel-leq-1*: *sel e ≤ 1*
**and** *pos-cards*: *x ∈ verts G ⟹ cf x > 0*

**begin**

## 5.1 Function for Join Trees and Selectivities

**definition** *matching-sel* :: *'a selectivity* ⇒ *bool* **where**
*matching-sel f* = (∀ *x y*.
(∃ *e*. (*tail G e*) = *x* ∧ (*head G e*) = *y* ∧ *f x y* = *sel e*)
∨ ((∄ *e*. (*tail G e*) = *x* ∧ (*head G e*) = *y*) ∧ *f x y* = *1*))

**definition** *match-sel* :: *'a selectivity* **where**
*match-sel x y* =
(*if* ∃ *e* ∈ *arcs G*. (*tail G e*) = *x* ∧ (*head G e*) = *y*
*then sel* (*THE e*. *e* ∈ *arcs G* ∧ (*tail G e*) = *x* ∧ (*head G e*) = *y*) *else 1*)

**definition** *matching-rels* :: *'a joinTree* ⇒ *bool* **where**
*matching-rels t* = (*relations t* ⊆ *verts G*)

**definition** *remove-sel* :: *'a* ⇒ *'b weight-fun* **where**
*remove-sel x* = (λ*b*. *if b*∈{*a* ∈ *arcs G*. *tail G a* = *x* ∨ *head G a* = *x*} *then 1 else sel b*)

**definition** *valid-tree* :: *'a joinTree* ⇒ *bool* **where**
*valid-tree t* = (*relations t* = *verts G* ∧ *distinct-relations t*)

**fun** *no-cross-products* :: *'a joinTree* ⇒ *bool* **where**
*no-cross-products* (*Relation rel*) = *True*
| *no-cross-products* (*Join l r*) = ((∃ *x*∈*relations l*. ∃ *y*∈*relations r*. *x* →$_G$ *y*)
∧ *no-cross-products l* ∧ *no-cross-products r*)

## 5.2 Proofs

Proofs that a query graph satisifies basic properties of join trees and selectivities.

**lemma** *sel-less-arc*: *sel x* < *1* ⟹ *x* ∈ *arcs G*
**using** *not-arc-sel-1* **by** *force*

**lemma** *joinTree-card-pos*: *matching-rels t* ⟹ *pos-rel-cards cf t*
**by**(*induction t*) (*auto simp*: *pos-cards pos-rel-cards-def matching-rels-def*)

**lemma** *symmetric-arcs*: *x*∈*arcs G* ⟹ ∃ *y*. *head G x* = *tail G y* ∧ *tail G x* = *head G y*
**using** *sym-arcs symmetric-conv* **by** *fast*

**lemma** *arc-ends-eq-impl-sel-eq*: *head G x* = *head G y* ⟹ *tail G x* = *tail G y* ⟹ *sel x* = *sel y*
**using** *sel-sym symmetric-arcs not-arc-sel-1* **by** *metis*

**lemma** *arc-ends-eq-impl-arc-eq*:
⟦*e1* ∈ *arcs G*; *e2* ∈ *arcs G*; *head G e1* = *head G e2*; *tail G e1* = *tail G e2*⟧ ⟹ *e1* = *e2*
**using** *no-multi-alt* **by** *blast*

**lemma** *matching-sel-simp-if-not1*:
⟦*matching-sel sf*; *sf x y* ≠ *1*⟧ ⟹ ∃ *e* ∈ *arcs G*. *tail G e* = *x* ∧ *head G e* = *y* ∧ *sf x y* = *sel e*
**using** *not-arc-sel-1* **unfolding** *matching-sel-def* **by** *fastforce*

**lemma** *matching-sel-simp-if-arc*:
⟦*matching-sel sf*; *e* ∈ *arcs G*⟧ ⟹ *sf* (*tail G e*) (*head G e*) = *sel e*
**unfolding** *matching-sel-def* **by** (*metis arc-ends-eq-impl-sel-eq*)

**lemma** *matching-sel1-if-no-arc*: *matching-sel sf* ⟹ ¬(*x* →$_G$ *y* ∨ *y* →$_G$ *x*) ⟹ *sf x y* = *1*
**using** *not-arc-sel-1* **unfolding** *arcs-ends-def arc-to-ends-def matching-sel-def image-iff* **by** *metis*

**lemma** *matching-sel-alt-aux1*:
*matching-sel f*
⟹ (∀ *x y*. (∃ *e* ∈ *arcs G*. (*tail G e*) = *x* ∧ (*head G e*) = *y* ∧ *f x y* = *sel e*)
∨ ((∄ *e*. *e* ∈ *arcs G* ∧ (*tail G e*) = *x* ∧ (*head G e*) = *y*) ∧ *f x y* = *1*))
**by** (*metis matching-sel-def arc-ends-eq-impl-sel-eq not-arc-sel-1*)

**lemma** *matching-sel-alt-aux2*:
(∀ *x y*.(∃ *e* ∈ *arcs G*. (*tail G e*) = *x* ∧ (*head G e*) = *y* ∧ *f x y* = *sel e*)
∨ ((∄ *e*. *e* ∈ *arcs G* ∧ (*tail G e*) = *x* ∧ (*head G e*) = *y*) ∧ *f x y* = *1*))
⟹ *matching-sel f*
**by** (*fastforce simp*: *not-arc-sel-1 matching-sel-def*)

**lemma** *matching-sel-alt*:
*matching-sel f*
= (∀ *x y*. (∃ *e* ∈ *arcs G*. (*tail G e*) = *x* ∧ (*head G e*) = *y* ∧ *f x y* = *sel e*)
∨ ((∄ *e*. *e* ∈ *arcs G* ∧ (*tail G e*) = *x* ∧ (*head G e*) = *y*) ∧ *f x y* = *1*))
**using** *matching-sel-alt-aux1 matching-sel-alt-aux2* **by** *blast*

**lemma** *matching-sel-symm*:
**assumes** *matching-sel f*
**shows** *sel-symm f*
**unfolding** *sel-symm-def*
**proof** (*standard*, *standard*)
**fix** *x y*
**show** *f x y* = *f y x*
**proof**(*cases* ∃ *e*∈*arcs G*. (*head G e*) = *x* ∧ (*tail G e*) = *y*)
**case** *True*
**then show** *?thesis* **using** *assms symmetric-arcs sel-sym* **unfolding** *matching-sel-def* **by** *metis*
**next**
**case** *False*
**then show** *?thesis* **by** (*metis assms symmetric-arcs matching-sel-def not-arc-sel-1 sel-sym*)
**qed**

**qed**

**lemma** *matching-sel-reasonable*: *matching-sel f* $\Longrightarrow$ *sel-reasonable f*
**using** *sel-reasonable-def matching-sel-def sel-pos sel-leq-1*
**by** (*metis le-numeral-extra*(*4*) *less-numeral-extra*(*1*))

**lemma** *matching-reasonable-cards*:
$[\![$*matching-sel f*; *matching-rels t*$]\!] \Longrightarrow$ *reasonable-cards cf f t*
**by** (*simp add*: *joinTree-card-pos matching-sel-reasonable pos-sel-reason-impl-reason*)

**lemma** *matching-sel-unique-aux*:
**assumes** *matching-sel f matching-sel g*
**shows** *f x y* = *g x y*
**proof**(*cases* $\exists$ *e*. *tail G e* = *x* $\wedge$ *head G e* = *y*)
**case** *True*
**then show** *?thesis*
**using** *assms arc-ends-eq-impl-sel-eq* **unfolding** *matching-sel-def* **by** *metis*
**next**
**case** *False*
**then show** *?thesis* **using** *assms* **unfolding** *matching-sel-def* **by** *fastforce*
**qed**

**lemma** *matching-sel-unique*: $[\![$*matching-sel f*; *matching-sel g*$]\!] \Longrightarrow$ *f* = *g*
**using** *matching-sel-unique-aux* **by** *blast*

**lemma** *match-sel-matching*[*intro*]: *matching-sel match-sel*
**unfolding** *matching-sel-alt*
**proof**(*standard*,*standard*)
**fix** *x y*
**show** ($\exists$ *e*$\in$*arcs G*. *tail G e* = *x* $\wedge$ *head G e* = *y* $\wedge$ *match-sel x y* = *sel e*) $\vee$
(($\nexists$ *e*. *e* $\in$ *arcs G* $\wedge$ *tail G e* = *x* $\wedge$ *head G e* = *y*) $\wedge$ *match-sel x y* = *1*)
**proof**(*cases* $\exists$ *e* $\in$ *arcs G*. *tail G e* = *x* $\wedge$ *head G e* = *y*)
**case** *True*
**then obtain** *e* **where** *e-def*: *e* $\in$ *arcs G tail G e* = *x head G e* = *y* **by** *blast*
**then have** *match-sel x y* = *sel* (*THE e*. *e* $\in$ *arcs G* $\wedge$ *tail G e* = *x* $\wedge$ *head G e* = *y*)
**unfolding** *match-sel-def* **by** *auto*
**moreover have** (*THE e*. *e* $\in$ *arcs G* $\wedge$ *tail G e* = *x* $\wedge$ *head G e* = *y*) = *e*
**using** *e-def arc-ends-eq-impl-arc-eq* **by** *blast*
**ultimately show** *?thesis* **using** *e-def* **by** *blast*
**next**
**case** *False*
**then show** *?thesis* **unfolding** *match-sel-def* **by** *auto*
**qed**
**qed**

**corollary** *match-sel-unique*: *matching-sel f* $\Longrightarrow$ *f* = *match-sel*
**using** *matching-sel-unique* **by** *blast*

**corollary** *match-sel1-if-no-arc*: $\neg(x \rightarrow_G y \vee y \rightarrow_G x) \Longrightarrow$ *match-sel x y = 1*
**using** *matching-sel1-if-no-arc* **by** *blast*

**corollary** *match-sel-symm*[*intro*]: *sel-symm match-sel*
**using** *matching-sel-symm* **by** *blast*

**corollary** *match-sel-reasonable*[*intro*]: *sel-reasonable match-sel*
**using** *matching-sel-reasonable* **by** *blast*

**corollary** *match-reasonable-cards*: *matching-rels t* $\Longrightarrow$ *reasonable-cards cf match-sel t*
**using** *matching-reasonable-cards* **by** *blast*

**lemma** *matching-rels-trans*: *matching-rels* (*Join l r*) = (*matching-rels l* $\wedge$ *matching-rels r*)
**using** *matching-rels-def* **by** *simp*

**lemma** *first-node-in-verts-if-rels-eq-verts*: *relations t = verts G* $\Longrightarrow$ *first-node t* $\in$ *verts G*
**unfolding** *first-node-eq-hd* **using** *inorder-eq-set hd-in-set*[*OF inorder-nempty*] **by** *fast*

**lemma** *first-node-in-verts-if-valid*: *valid-tree t* $\Longrightarrow$ *first-node t* $\in$ *verts G*
**using** *first-node-in-verts-if-rels-eq-verts valid-tree-def* **by** *simp*

**lemma** *dominates-sym*: ($x \rightarrow_G y$) $\longleftrightarrow$ ($y \rightarrow_G x$)
**using** *graph-symmetric* **by** *blast*

**lemma** *no-cross-mirror-eq*: *no-cross-products* (*mirror t*) = *no-cross-products t*
**using** *graph-symmetric* **by**(*induction t*) *auto*

**lemma** *no-cross-create-ldeep-rev-app*:
⟦*ys*≠[]; *no-cross-products* (*create-ldeep-rev* (*xs*@*ys*))⟧ $\Longrightarrow$ *no-cross-products* (*create-ldeep-rev ys*)
**proof**(*induction xs*@*ys arbitrary*: *xs rule*: *create-ldeep-rev.induct*)
**case** (*2 x*)
**then show** *?case* **by** (*metis append-eq-Cons-conv append-is-Nil-conv*)
**next**
**case** (*3 x y zs*)
**then show** *?case*
**proof**(*cases xs*)
**case** *Nil*
**then show** *?thesis* **using** *3.prems*(*2*) **by** *simp*
**next**
**case** (*Cons x′ xs′*)
**have** *no-cross-products* (*Join* (*create-ldeep-rev* (*y*#*zs*)) (*Relation x*))
**using** *3.hyps*(*2*) *3.prems*(*2*) *create-ldeep-rev.simps*(*3*)[*of x y zs*] **by** *simp*
**then have** *no-cross-products* (*create-ldeep-rev* (*y*#*zs*)) **by** *simp*
**then show** *?thesis* **using** *3.hyps 3.prems*(*1*) *Cons* **by** *simp*

**qed**
**qed**(*simp*)

**lemma** *no-cross-create-ldeep-app*:
⟦*xs*≠[]; *no-cross-products* (*create-ldeep* (*xs*@*ys*))⟧ ⟹ *no-cross-products* (*create-ldeep* *xs*)
**by** (*simp add*: *create-ldeep-def no-cross-create-ldeep-rev-app*)

**lemma** *matching-rels-if-no-cross*: ⟦∀ *r*. *t* ≠ *Relation r*; *no-cross-products t*⟧ ⟹ *matching-rels t*
**unfolding** *matching-rels-def* **by**(*induction t*) *fastforce*+

**lemma** *no-cross-awalk*:
⟦*matching-rels t*; *no-cross-products t*; *x* ∈ *relations t*; *y* ∈ *relations t*⟧
⟹ ∃ *p*. *awalk x p y* ∧ *set* (*awalk-verts x p*) ⊆ *relations t*
**proof**(*induction t arbitrary*: *x y*)
**case** (*Relation rel*)
**then have** *x* ∈ *verts G* **using** *matching-rels-def* **by** *blast*
**then have** *awalk x* [] *x* **by** (*simp add*: *awalk-Nil-iff*)
**then show** *?case* **using** *Relation*(*3*,*4*) **by** *force*
**next**
**case** (*Join l r*)
**then consider** *x* ∈ *relations l y* ∈ *relations l* | *x* ∈ *relations r y* ∈ *relations l*
| *x* ∈ *relations l y* ∈ *relations r* | *x* ∈ *relations r y* ∈ *relations r*
**by** *force*
**then show** *?case*
**proof**(*cases*)
**case** *1*
**then show** *?thesis* **using** *Join.IH*(*1*)[*of x y*] *Join.prems*(*1*,*2*) *matching-rels-trans*
**by** *auto*
**next**
**case** *2*
**then obtain** *x′ y′ e* **where** *e-def*:
*x′* ∈ *relations r y′* ∈ *relations l tail G e* = *y′ head G e* = *x′ e* ∈ *arcs G*
**using** *Join.prems*(*2*) **by** *auto*
**then obtain** *e2* **where** *e2-def*: *tail G e2* = *x′ head G e2* = *y′ e2* ∈ *arcs G*
**using** *symmetric-conv* **by** *force*
**obtain** *p1* **where** *p1-def*: *awalk y′ p1 y* ∧ *set* (*awalk-verts y′ p1*) ⊆ *relations l*
**using** *Join.IH*(*1*) *Join.prems*(*1*,*2*) *2*(*2*) *matching-rels-trans e-def*(*2*) **by** *fastforce*
**obtain** *p2* **where** *p2-def*: *awalk x p2 x′* ∧ *set* (*awalk-verts x p2*) ⊆ *relations r*
**using** *Join.IH*(*2*) *Join.prems*(*1*,*2*) *2*(*1*) *matching-rels-trans e-def*(*1*) **by** *fastforce*
**have** *awalk x* (*p2*@[*e2*]@*p1*) *y*
**using** *e2-def p1-def p2-def awalk-appendI arc-implies-awalk* **by** *blast*
**moreover from** *this* **have** *set* (*awalk-verts x* (*p2*@[*e2*]@*p1*)) ⊆ *relations* (*Join l r*)
**using** *p1-def p2-def awalk-verts-append3* **by** *auto*
**ultimately show** *?thesis* **by** *blast*

**next**
**case** *3*
**then obtain** $x'$ $y'$ $e$ **where** *e-def*:
$x' \in$ *relations* $l$ $y' \in$ *relations* $r$ *tail* $G$ $e = x'$ *head* $G$ $e = y'$ $e \in$ *arcs* $G$
**using** *Join.prems*(*2*) **by** *auto*
**obtain** *p1* **where** *p1-def*: *awalk* $y'$ *p1* $y \wedge$ *set* (*awalk-verts* $y'$ *p1*) $\subseteq$ *relations* $r$
**using** *Join.IH*(*2*) *Join.prems*(*1*,*2*) *3*(*2*) *matching-rels-trans e-def*(*2*) **by** *fastforce*
**obtain** *p2* **where** *p2-def*: *awalk* $x$ *p2* $x' \wedge$ *set* (*awalk-verts* $x$ *p2*) $\subseteq$ *relations* $l$
**using** *Join.IH*(*1*) *Join.prems*(*1*,*2*) *3*(*1*) *matching-rels-trans e-def*(*1*) **by** *fastforce*
**have** *awalk* $x$ (*p2*@[*e*]@*p1*) $y$
**using** *e-def*(*3*−*5*) *p1-def p2-def awalk-appendI arc-implies-awalk* **by** *blast*
**moreover from** *this* **have** *set* (*awalk-verts* $x$ (*p2*@[*e*]@*p1*)) $\subseteq$ *relations* (*Join* $l$ $r$)
**using** *p1-def p2-def awalk-verts-append3* **by** *auto*
**ultimately show** *?thesis* **by** *blast*
**next**
**case** *4*
**then show** *?thesis* **using** *Join.IH*(*2*)[*of* $x$ $y$] *Join.prems*(*1*,*2*) *matching-rels-trans* **by** *auto*
**qed**
**qed**

**lemma** *no-cross-apath*:
⟦*matching-rels t*; *no-cross-products t*; $x \in$ *relations t*; $y \in$ *relations t*⟧
$\Longrightarrow \exists p.$ *apath* $x$ $p$ $y \wedge$ *set* (*awalk-verts* $x$ $p$) $\subseteq$ *relations t*
**using** *no-cross-awalk apath-awalk-to-apath awalk-to-apath-verts-subset* **by** *blast*

**lemma** *no-cross-reachable*:
⟦*matching-rels t*; *no-cross-products t*; $x \in$ *relations t*; $y \in$ *relations t*⟧ $\Longrightarrow x \rightarrow^* y$
**using** *no-cross-awalk reachable-awalk* **by** *blast*

**corollary** *reachable-if-no-cross*:
⟦$\exists t.$ *relations* $t$ = *verts* $G \wedge$ *no-cross-products t*; $x \in$ *verts* $G$; $y \in$ *verts* $G$⟧ $\Longrightarrow$ $x \rightarrow^* y$
**using** *no-cross-reachable matching-rels-def* **by** *blast*

**lemma** *remove-sel-sym*:
⟦*tail* $G$ $e_1$ = *head* $G$ $e_2$; *head* $G$ $e_1$ = *tail* $G$ $e_2$⟧ $\Longrightarrow$ (*remove-sel* $x$) $e_1$ = (*remove-sel* $x$) $e_2$
**by**(*metis* (*no-types*, *lifting*) *mem-Collect-eq not-arc-sel-1 remove-sel-def sel-sym*)+

**lemma** *remove-sel-1*: $e \notin$ *arcs* $G \Longrightarrow$ (*remove-sel* $x$) $e = 1$
**apply**(*cases* $e \in \{a \in$ *arcs* $G.$ *tail* $G$ $a = x \vee$ *head* $G$ $a = x\}$)
**by**(*auto simp*: *not-arc-sel-1 sel-sym remove-sel-def*)

**lemma** *del-vert-remove-sel-1*:
**assumes** $e \notin$ *arcs* ((*del-vert* $x$))

**shows** (*remove-sel x*) *e* = *1*
**proof**(*cases e*∈{*a* ∈ *arcs G*. *tail G a* = *x* ∨ *head G a* = *x*})
**case** *True*
**then show** *?thesis* **by** (*simp add*: *remove-sel-def*)
**next**
**case** *False*
**then have** *e* ∉ *arcs G* **using** *assms arcs-del-vert* **by** *simp*
**then show** *?thesis* **using** *remove-sel-def not-arc-sel-1* **by** *simp*
**qed**

**lemma** *remove-sel-pos*: *remove-sel x e* > *0*
**by**(*cases e*∈{*a* ∈ *arcs G*. *tail G a* = *x* ∨ *head G a* = *x*}) (*auto simp*: *remove-sel-def sel-pos*)

**lemma** *remove-sel-leq-1*: *remove-sel x e* ≤ *1*
**by**(*cases e*∈{*a* ∈ *arcs G*. *tail G a* = *x* ∨ *head G a* = *x*}) (*auto simp*: *remove-sel-def sel-leq-1*)

**lemma** *del-vert-pos-cards*: *x* ∈ *verts* (*del-vert y*) ⟹ *cf x* > *0*
**by**(*cases x*=*y*) (*auto simp*: *remove-sel-def del-vert-def pos-cards*)

**lemma** *del-vert-remove-sel-query-graph*:
*query-graph G sel cf* ⟹ *query-graph* (*del-vert x*) (*remove-sel x*) *cf*
**by** (*simp add*: *del-vert-pos-cards del-vert-remove-sel-1 graph-del-vert remove-sel-sym remove-sel-leq-1 remove-sel-pos query-graph.intro graph-axioms head-del-vert query-graph-axioms-def tail-del-vert*)

**lemma** *finite-nempty-set-min*:
**assumes** *xs* ≠ {} **and** *finite xs*
**shows** ∃ *x*. *min-degree xs x*
**proof** −
**have** *finite xs* **using** *assms*(*2*) **by** *simp*
**then show** *?thesis*
**using** *assms* **proof** (*induction xs rule*: *finite-induct*)
**case** *empty*
**then show** *?case* **by** *simp*
**next**
**case** *ind*: (*insert x xs*)
**then show** *?case*
**proof**(*cases xs*)
**case** *emptyI*
**then show** *?thesis* **by** (*metis order-refl singletonD singletonI*)
**next**
**case** (*insertI xs*′ *x*′)
**then have** ∃ *a*. *min-degree xs a* **using** *ind* **by** *simp*
**then show** *?thesis*
**using** *ind* **by** (*metis order-trans insert-iff le-cases*)
**qed**
**qed**

**qed**

**lemma** *no-cross-reachable-graph'*:
  $[\![\exists\, t.\ relations\ t = verts\ G \wedge no\text{-}cross\text{-}products\ t;\ x \in verts\ G;\ y \in verts\ G]\!]$
    $\Longrightarrow x \rightarrow^*_{mk\text{-}symmetric\ G} y$
  **by** (*simp add*: *reachable-mk-symmetricI reachable-if-no-cross*)

**lemma** *verts-nempty-if-tree*: $\exists\, t.\ relations\ t \subseteq verts\ G \Longrightarrow verts\ G \neq \{\}$
  **using** *relations-nempty* **by** *fast*

**lemma** *connected-if-tree*: $\exists\, t.\ relations\ t = verts\ G \wedge no\text{-}cross\text{-}products\ t \Longrightarrow$ *connected G*
 **using** *no-cross-reachable-graph' connected-def strongly-connected-def verts-nempty-if-tree*
  **by** *fastforce*

**end**

**locale** *nempty-query-graph = query-graph* +
  **assumes** *non-empty*: $verts\ G \neq \{\}$

## 5.3 Pair Query Graph

Alternative definition based on pair graphs

**locale** *pair-query-graph = pair-graph* +
  **fixes** *sel* :: $('a \times 'a)$ *weight-fun*
  **fixes** *cf* :: $'a \Rightarrow real$
  **assumes** *sel-sym*: $[\![tail\ G\ e_1 = head\ G\ e_2;\ head\ G\ e_1 = tail\ G\ e_2]\!] \Longrightarrow sel\ e_1 = sel\ e_2$
      **and** *not-arc-sel-1*: $e \notin parcs\ G \Longrightarrow sel\ e = 1$
      **and** *sel-pos*: $sel\ e > 0$
      **and** *sel-leq-1*: $sel\ e \leq 1$
      **and** *pos-cards*: $x \in pverts\ G \Longrightarrow cf\ x > 0$

**sublocale** *pair-query-graph* $\subseteq$ *query-graph*
  **by**(*unfold-locales*) (*auto simp*: *sel-sym not-arc-sel-1 sel-pos sel-leq-1 pos-cards*)

**context** *pair-query-graph*
**begin**

**lemma** *matching-sel f* $\longleftrightarrow$ $(\forall\, x\ y.\ sel\ (x,y) = f\ x\ y)$
  **using** *matching-sel-def sel-sym* **by** *fastforce*

**end**

**end**

**theory** *Directed-Tree-Additions*
  **imports** *Graph-Additions Shortest-Path-Tree*

**begin**

# 6 Directed Tree Additions

**context** *directed-tree*
**begin**

**lemma** *reachable1-not-reverse*: $x \rightarrow^+_T y \Longrightarrow \neg\ y \rightarrow^+_T x$
**by** (*metis awalk-Nil-iff reachable1-awalk reachable1-in-verts*(2) *trancl-trans unique-awalk-All*)

**lemma** *in-arcs-root*: *in-arcs T root* = {}
**using** *in-degree-root-zero* **by** (*auto simp*: *in-degree-def in-arcs-finite root-in-T*)

**lemma** *dominated-not-root*: $u \rightarrow_T v \Longrightarrow v \neq root$
**using** *adj-in-verts*(1) *reachable1-not-reverse reachable-from-root* **by** *blast*

**lemma** *dominated-notin-awalk*: $[\![u \rightarrow_T v;\ awalk\ r\ p\ u]\!] \Longrightarrow v \notin set\ (awalk\text{-}verts\ r\ p)$
**using** *awalk-verts-reachable-to reachable1-not-reverse* **by** *blast*

**lemma** *apath-if-awalk*: *awalk r p v* $\Longrightarrow$ *apath r p v*
**using** *apath-def awalk-cyc-decompE′ closed-w-imp-cycle cycle-free* **by** *blast*

**lemma** *awalk-verts-arc1-app*: *tail T e* $\in$ *set* (*awalk-verts r* (*p1*@*e*#*p2*))
**using** *awalk-verts-arc1* **by** *auto*

**lemma** *apath-over-inarc-if-dominated*:
**assumes** $u \rightarrow_T v$
**shows** $\exists p.$ *apath root p v* $\wedge$ $u \in$ *set* (*awalk-verts root p*)
**proof** −
**obtain** *p* **where** *p-def*: *awalk root p u* **using** *assms unique-awalk* **by** *force*
**obtain** *e* **where** *e-def*: *e* $\in$ *arcs T tail T e* = *u head T e* = *v* **using** *assms* **by** *blast*
**then have** *awalk root* (*p*@[*e*]) *v* **using** *p-def arc-implies-awalk* **by** *auto*
**then show** *?thesis* **using** *apath-if-awalk e-def*(2) *awalk-verts-arc1-app* **by** *blast*
**qed**

**end**

**locale** *finite-directed-tree* = *directed-tree* + *fin-digraph T*

Undirected, connected graphs are acyclic iff the number of edges is |verts| - 1. Since undirected graphs are modelled as bidirected graphs the number of edges is doubled.

**locale** *undirected-tree* = *graph* +
**assumes** *connected*: *connected G*
**and** *acyclic*: *card* (*arcs G*) $\leq$ 2 $*$ (*card* (*verts G*) − 1)

## 6.1 Directed Trees of Connected Trees

### 6.1.1 Tranformation using BFS

Assumes existence of a conversion function (like BFS) that contains all reachable vertices.

**locale** *bfs-tree* = *directed-tree T root* + *subgraph T G* **for** *G T root* +
  **assumes** *root-in-G*: *root* ∈ *verts G*
    **and** *all-reachables*: *verts T* = {*v*. *root* →**G v*}
**begin**

**lemma** *dom-in-G*: *u* →*T v* ⟹ *u* →*G v*
  **by** (*simp add*: *G.adj-mono sub-G*)

**lemma** *tailT-eq-tailG*: *tail T* = *tail G*
  **using** *sub-G* **by** (*simp add*: *Digraph-Component.subgraph-def compatible-def*)

**lemma** *headT-eq-headG*: *head T* = *head G*
  **using** *sub-G* **by** (*simp add*: *Digraph-Component.subgraph-def compatible-def*)

**lemma** *verts-T-subset-G*: *verts T* ⊆ *verts G*
  **by** (*metis awalk-sub-imp-awalk G.awalk-last-in-verts subsetI unique-awalk*)

**lemma** *reachable-verts-G-subset-T*:
  ∀ *x*∈*verts G*. *root* →**G x* ⟹ *verts T* ⊇ *verts G*
  **using** *all-reachables* **by** (*simp add*: *subset-eq*)

**lemma** *reachable-verts-G-eq-T*: ∀ *x*∈*verts G*. *root* →**G x* ⟹ *verts T* = *verts G*
  **by** (*simp add*: *reachable-verts-G-subset-T set-eq-subset verts-T-subset-G*)

**lemma** *connected-verts-G-eq-T*:
  **assumes** *graph G* **and** *connected G*
  **shows** *verts T* = *verts G*
**proof** −
  **have** *root* ∈ *verts G* **using** *root-in-G* **by** *fast*
 **then have** ∀ *x*∈*verts G*. *root* →**G x* **using** *graph.connected-iff-reachable assms*(*1*,*2*)
**by** *blast*
  **then show** *?thesis* **using** *reachable-verts-G-eq-T* **by** *blast*
**qed**

**lemma** *Suc-card-if-fin*: *fin-digraph G* ⟹ ∃ *n*. *Suc n* = *card* (*verts G*)
 **using** *root-in-G card-0-eq not0-implies-Suc*[*of card* (*verts G*)] *fin-digraph.finite-verts*
**by** *force*

**corollary** *Suc-card-if-graph*: *graph G* ⟹ ∃ *n*. *Suc n* = *card* (*verts G*)
  **using** *Suc-card-if-fin graph.axioms*(*1*) *digraph.axioms*(*1*) **by** *blast*

**lemma** *con-Suc-card-arcs-eq-card-verts*:
  ⟦*graph G*; *connected G*⟧ ⟹ *Suc* (*card* (*arcs T*)) = *card* (*verts G*)

**using** *Suc-card-arcs-eq-card-verts connected-verts-G-eq-T Suc-card-if-graph* **by** *fastforce*

**lemma** *reverse-arc-in-G*:
**assumes** *graph G* **and** *e1* ∈ *arcs T*
**shows** ∃ *e2* ∈ *arcs G*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*
**proof** −
**interpret** *graph G* **using** *assms*(*1*) **.**
**have** *e1* ∈ *arcs G* **using** *assms*(*2*) *sub-G* **by** *blast*
**then show** *?thesis* **using** *sym-arcs symmetric-conv* **by** *fastforce*
**qed**

**lemma** *reverse-arc-notin-T*:
**assumes** *e1* ∈ *arcs T* **and** *head G e2* = *tail G e1* **and** *head G e1* = *tail G e2*
**shows** *e2* ∉ *arcs T*
**proof**
**assume** *asm*: *e2* ∈ *arcs T*
**then have** *tail T e2* $\rightarrow_T$ *head T e2* **by** (*simp add*: *in-arcs-imp-in-arcs-ends*)
**then have** *head G e1* $\rightarrow_T$ *tail G e1*
**using** *assms*(*2*,*3*) *sub-G* **by**(*simp add*: *Digraph-Component.subgraph-def compatible-def*)
**moreover have** *tail G e1* $\rightarrow_T$ *head G e1*
**using** *assms*(*1*) *sub-G*
**by**(*simp add*: *Digraph-Component.subgraph-def compatible-def in-arcs-imp-in-arcs-ends*)
**ultimately show** *False* **using** *reachable1-not-reverse* **by** *blast*
**qed**

**lemma** *reverse-arc-in-G-only*:
**assumes** *graph G* **and** *e1* ∈ *arcs T*
**shows** ∃ *e2* ∈ *arcs G*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2* ∧ *e2* ∉ *arcs T*
**using** *reverse-arc-in-G reverse-arc-notin-T assms* **by** *blast*

**lemma** *no-multi-T-G*:
**assumes** *e1* ∈ *arcs T* **and** *e2* ∈ *arcs T* **and** *e1* ≠ *e2*
**shows** *head G e1* ≠ *head G e2* ∨ *tail G e1* ≠ *tail G e2*
**using** *nomulti.no-multi-arcs assms sub-G*
**by**(*auto simp*: *Digraph-Component.subgraph-def compatible-def arc-to-ends-def*)

**lemma** *T-arcs-compl-fin*:
**assumes** *fin-digraph G* **and** *es* ⊆ *arcs T*
**shows** *finite* {*e2*∈ *arcs G*. (∃ *e1* ∈ *es*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*)}
**using** *assms fin-digraph.finite-arcs* **by** *fastforce*

**corollary** *T-arcs-compl-fin′*:
**assumes** *graph G* **and** *es* ⊆ *arcs T*
**shows** *finite* {*e2*∈ *arcs G*. (∃ *e1* ∈ *es*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*)}

**using** *assms T-arcs-compl-fin graph.axioms(1) digraph.axioms(1)* **by** *blast*

**lemma** *fin-verts-T*: *fin-digraph G* $\Longrightarrow$ *finite* (*verts T*)
**using** *fin-digraph.finite-verts finite-subset verts-T-subset-G* **by** *auto*

**corollary** *fin-verts-T'*: *graph G* $\Longrightarrow$ *finite* (*verts T*)
**using** *fin-verts-T graph.axioms(1) digraph.axioms(1)* **by** *blast*

**lemma** *fin-arcs-T*: *fin-digraph G* $\Longrightarrow$ *finite* (*arcs T*)
**using** *fin-verts-T verts-finite-imp-arcs-finite* **by** *auto*

**corollary** *fin-arcs-T'*: *graph G* $\Longrightarrow$ *finite* (*arcs T*)
**using** *fin-arcs-T graph.axioms(1) digraph.axioms(1)* **by** *blast*

**lemma** *T-arcs-compl-card-eq*:
**assumes** *graph G* **and** *es* $\subseteq$ *arcs T*
**shows** *card* {*e2*$\in$ *arcs G*. ($\exists$ *e1* $\in$ *es*. *head G e2* = *tail G e1* $\wedge$ *head G e1* = *tail G e2*)} = *card es*
**using** *finite-subset*[*OF assms*(*2*) *fin-arcs-T'*[*OF assms*(*1*)]] *assms*
**proof**(*induction es rule*: *finite-induct*)
**case** (*insert e1 es*)
**let** *?ees* = {*e2* $\in$ *arcs G*. $\exists$ *e1*$\in$*insert e1 es*. *head G e2* = *tail G e1* $\wedge$ *head G e1* = *tail G e2*}
**let** *?es* = {*e2* $\in$ *arcs G*. $\exists$ *e1*$\in$*es*. *head G e2* = *tail G e1* $\wedge$ *head G e1* = *tail G e2*}
**obtain** *e2* **where** *e2-def*: *e2* $\in$ *arcs G head G e2* = *tail G e1 head G e1* = *tail G e2*
**using** *reverse-arc-in-G-only insert.prems* **by** *blast*
**then have** *e2-notin*: *e2* $\notin$ {*e2* $\in$ *arcs G*. $\exists$ *e1*$\in$*es*. *head G e2* = *tail G e1* $\wedge$ *head G e1* = *tail G e2*}
**using** *insert.hyps*(*2*) *insert.prems*(*2*) *no-multi-T-G* **by** *fastforce*
**have** $\forall$ *e3* $\in$ *arcs G*. *e2* = *e3* $\vee$ *head G e3* $\neq$ *head G e2* $\vee$ *tail G e3* $\neq$ *tail G e2*
**using** *e2-def*(*1*) *nomulti-digraph.no-multi-alt digraph.axioms*(*3*) *graph.axioms*(*1*) *insert.prems*(*1*)
**by** *fast*
**then have** *?ees* = *insert e2 ?es* **using** *e2-def* **by** *auto*
**moreover have** *finite ?es* **using** *insert.prems T-arcs-compl-fin'* **by** *simp*
**ultimately have** *card ?ees* = *Suc* (*card ?es*) **using** *e2-notin* **by** *simp*
**then show** *?case* **using** *insert* **by** *force*
**qed**(*simp*)

**lemma** *arcs-graph-G-ge-2vertsT*:
**assumes** *graph G*
**shows** *card* (*arcs G*) $\geq$ *2* $*$ (*card* (*verts T*) $-$ *1*)
**proof** $-$
**let** *?compl* = {*e2*$\in$ *arcs G*. ($\exists$ *e1* $\in$ *arcs T*. *head G e2* = *tail G e1* $\wedge$ *head G e1* = *tail G e2*)}
**interpret** *graph G* **by** (*rule assms*)
**have** $\forall$ *e1* $\in$ *arcs T*. $\exists$ *e2* $\in$ *arcs G*. *head G e2* = *tail G e1* $\wedge$ *head G e1* = *tail*

*G e2*
**using** *reverse-arc-in-G-only assms* **by** *blast*
**have** *fin1*: *finite ?compl* **by** *simp*
**have** *?compl* ∩ *arcs T* = {} **using** *reverse-arc-notin-T* **by** *blast*
**then have** *card* (*?compl* ∪ *arcs T*) = *card ?compl* + *card* (*arcs T*)
**using** *card-Un-disjoint*[*OF fin1 fin-arcs-T'*] **by** *blast*
**moreover have** *?compl* ∪ *arcs T* ⊆ *arcs G* **using** *sub-G* **by** *blast*
**moreover have** *finite* (*arcs G*) **by** *simp*
**ultimately have** *card ?compl* + *card* (*arcs T*) ≤ *card* (*arcs G*)
**using** *card-mono*[*of arcs G ?compl* ∪ *arcs T*] **by** *presburger*
**moreover have** *card* (*arcs T*) = (*card* (*verts T*) − *1*)
**using** *Suc-card-arcs-eq-card-verts assms* **by** (*simp add*: *fin-verts-T'*)
**ultimately show** *?thesis* **using** *T-arcs-compl-card-eq* **by** *fastforce*
**qed**

**lemma** *arcs-graph-G-ge-2vertsG*:
⟦*graph G*; *connected G*⟧ ⟹ *card* (*arcs G*) ≥ *2* ∗ (*card* (*verts G*) − *1*)
**using** *arcs-graph-G-ge-2vertsT connected-verts-G-eq-T* **by** *simp*

**lemma** *arcs-undir-G-eq-2vertsG*:
⟦*undirected-tree G*⟧ ⟹ *card* (*arcs G*) = *2* ∗ (*card* (*verts G*) − *1*)
**using** *arcs-graph-G-ge-2vertsG undirected-tree.acyclic undirected-tree.axioms*(*1*)
*undirected-tree.connected* **by** *fastforce*

**lemma** *undir-arcs-compl-un-eq-arcs*:
**assumes** *undirected-tree G*
**shows** {*e2*∈ *arcs G*. (∃ *e1* ∈ *arcs T*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*)} ∪ *arcs T*
= *arcs G*
**proof** −
**let** *?compl* = {*e2*∈ *arcs G*. (∃ *e1* ∈ *arcs T*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*)}
**interpret** *undirected-tree G* **using** *assms*(*1*) *undirected-tree.axioms*(*1*) **by** *fast*
**have** *?compl* ∩ *arcs T* = {} **using** *reverse-arc-notin-T* **by** *blast*
**then have** *0*: *card* (*?compl* ∪ *arcs T*) = *card ?compl* + *card* (*arcs T*)
**by** (*simp add*: *card-Un-disjoint fin-arcs-T' graph-axioms*)
**have** *card* (*arcs T*) = (*card* (*verts T*) − *1*)
**using** *Suc-card-arcs-eq-card-verts* **by** (*simp add*: *fin-verts-T' graph-axioms*)
**then have** *card ?compl* + *card* (*arcs T*) = *2* ∗ (*card* (*verts G*) − *1*)
**using** *T-arcs-compl-card-eq connected-verts-G-eq-T connected* **by** *fastforce*
**moreover have** *card* (*arcs G*) = *2* ∗ (*card* (*verts G*) − *1*)
**using** *assms arcs-undir-G-eq-2vertsG* **by** *blast*
**moreover have** *?compl* ∪ *arcs T* ⊆ *arcs G* **using** *sub-G* **by** *blast*
**ultimately show** *?thesis* **by** (*simp add*: *0 card-subset-eq*)
**qed**

**lemma** *split-fst-nonelem*:
⟦¬*set xs* ⊆ *X*; *set xs* ⊆ *Y*⟧ ⟹ ∃ *x ys zs*. *ys*@*x*#*zs*=*xs* ∧ *x* ∉ *X* ∧ *x* ∈ *Y* ∧ *set ys* ⊆ *X*

**proof**(*induction xs*)
**case** (*Cons x xs*)
**then show** *?case*
**proof**(*cases x* ∈ *X*)
**case** *True*
**then obtain** *z ys zs* **where** *ys-def*: *ys@z#zs=xs z* ∉ *X z* ∈ *Y set ys* ⊆ *X* **using** *Cons* **by** *auto*
**then have** *set* (*x#ys*) ⊆ *X* **using** *True* **by** *simp*
**then show** *?thesis* **using** *ys-def*(*1−3*) *append-Cons* **by** *fast*
**next**
**case** *False*
**then show** *?thesis* **using** *Cons.prems*(*2*) **by** *fastforce*
**qed**
**qed**(*simp*)

**lemma** *source-no-inarc-T*: *head G e* = *root* ⟹ *e* ∉ *arcs T*
**using** *in-arcs-root sub-G* **by** (*auto simp*: *Digraph-Component.subgraph-def compatible-def*)

**lemma** *source-all-outarcs-T*:
⟦*undirected-tree G*; *tail G e* = *root*; *e* ∈ *arcs G*⟧ ⟹ *e* ∈ *arcs T*
**using** *source-no-inarc-T undir-arcs-compl-un-eq-arcs* **by** *blast*

**lemma** *cas-G-T*: *G.cas* = *cas*
**using** *sub-G compatible-cas* **by** *fastforce*

**lemma** *awalk-G-T*: *u* ∈ *verts T* ⟹ *set p* ⊆ *arcs T* ⟹ *G.awalk u p* = *awalk u p*
**using** *cas-G-T awalk-def G.awalk-def sub-G* **by** *fastforce*

**corollary** *awalk-G-T-root*: *set p* ⊆ *arcs T* ⟹ *G.awalk root p* = *awalk root p*
**using** *awalk-G-T root-in-T* **by** *blast*

**lemma** *awalk-verts-G-T*: *G.awalk-verts* = *awalk-verts*
**using** *sub-G compatible-awalk-verts* **by** *blast*

**lemma** *apath-sub-imp-apath*: *apath u p v* ⟹ *G.apath u p v*
**by** (*simp add*: *G.apath-def apath-def awalk-sub-imp-awalk awalk-verts-G-T*)

**lemma** *outarc-inT-if-head-not-inarc*:
**assumes** *undirected-tree G*
**and** *tail G e2* = *v* **and** *e2* ∈ *arcs G* **and** *head G e2* ≠ *u* **and** *u* →$_T$ *v*
**shows** *e2* ∈ *arcs T*
**proof** (*rule ccontr*)
**let** *?compl* = {*e2*∈ *arcs G*. (∃ *e1* ∈ *arcs T*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*)}
**assume** *e2* ∉ *arcs T*
**then have** *e2* ∈ *?compl* **using** *assms*(*3*) *undir-arcs-compl-un-eq-arcs*[*OF assms*(*1*)] **by** *blast*
**then obtain** *e1* **where** *e1-def*: *e1* ∈ *arcs T head G e2* = *tail T e1 head T e1*

$= v$
**using** *sub-G assms*(*2*) **by** (*auto simp*: *Digraph-Component.subgraph-def compatible-def*)
**obtain** *e* **where** $e \in arcs\ T\ tail\ T\ e = u\ head\ T\ e = v$ **using** *assms*(*5*) **by** *blast*
**then show** *False* **using** *two-in-arcs-contr e1-def assms*(*4*) **by** *blast*
**qed**

**corollary** *reverse-arc-if-out-arc-undir*:
$[\![$*undirected-tree G*; *tail G e2* $= v$; $e2 \in arcs\ G$; $e2 \notin arcs\ T$; $u \rightarrow_T v]\!] \Longrightarrow$ *head G e2* $= u$
**using** *outarc-inT-if-head-not-inarc* **by** *blast*

**lemma** *undir-path-in-dir*:
**assumes** *undirected-tree G G.apath root p v*
**shows** $set\ p \subseteq arcs\ T$
**proof** (*rule ccontr*)
**assume** *asm*: $\neg set\ p \subseteq arcs\ T$
**have** $set\ p \subseteq arcs\ G$ **using** *assms*(*2*) *G.apath-def G.awalk-def* **by** *fast*
**then obtain** *e p1 p2* **where** *e-def*: $p1\ @\ e\ \#\ p2 = p\ e \notin arcs\ T\ e \in arcs\ G\ set\ p1 \subseteq arcs\ T$
**using** *split-fst-nonelem*[*OF asm*, *of arcs G*] **by** *auto*
**show** *False*
**proof**(*cases p1*=[])
**case** *True*
**then have** *tail G e* $=$ *root* **using** *assms*(*2*) *e-def*(*1*) *G.apath-Cons-iff* **by** *auto*
**then show** *?thesis* **using** *source-all-outarcs-T*[*OF assms*(*1*)] *e-def*(*2*,*3*) **by** *blast*
**next**
**case** *False*
**then have** *awalk-G*: *G.awalk root* (*p1 @ e # p2*) *v*
**using** *assms*(*2*) *pre-digraph.apath-def e-def*(*1*) **by** *fast*
**then have** *G.awalk root p1* (*tail G e*) **by** *force*
**then have** *awalk-p1T*: *awalk root p1* (*tail T e*)
**using** *e-def*(*4*) *sub-G cas-G-T root-in-T*
**by** (*simp add*: *Digraph-Component.subgraph-def pre-digraph.awalk-def compatible-def*)
**then have** $root \rightarrow^+_T tail\ T\ e$ **using** *False reachable1-awalkI* **by** *auto*
**then obtain** *u* **where** *u-def*: $u \rightarrow_T tail\ T\ e$ **using** *tranclD2* **by** *metis*
**have** *tail T e* $=$ *tail G e*
**using** *sub-G* **by** (*simp add*: *Digraph-Component.subgraph-def compatible-def*)
**then have** *hd-e-u*: *head G e* $= u$
**using** *reverse-arc-if-out-arc-undir*[*OF assms*(*1*)] *u-def e-def*(*2*,*3*) **by** *simp*
**have** *head T* (*last p1*) $=$ *tail T e* **using** *False awalk-p1T awalk-verts-conv* **by** *fastforce*
**then have** *tail T* (*last p1*) $= u$
**using** *False u-def e-def*(*4*) *two-in-arcs-contr last-in-set* **by** *fastforce*
**then have** *0*: *tail G* (*last p1*) $= u$
**using** *sub-G* **by** (*simp add*: *Digraph-Component.subgraph-def compatible-def*)
**obtain** *ps* **where** *ps* @ [*last p1*] $=$ *p1* **using** *False append-butlast-last-id* **by**

```
auto
    then have ps-def: ps @ [last p1] @ e # p2 = p using e-def by auto
    then have awalk-G: G.awalk root (ps @ [last p1] @ e # p2) v
      using assms(2) by (simp add: pre-digraph.apath-def)
    have ¬(distinct (G.awalk-verts root p))
      using G.not-distinct-if-head-eq-tail[OF 0 hd-e-u awalk-G] ps-def by simp
    then show ?thesis using assms(2) G.apath-def by blast
  qed
qed

lemma source-reach-all: ⟦graph G; connected G; v ∈ verts G⟧ ⟹ root →*_G v
  by (simp add: graph.connected-iff-reachable root-in-G)

lemma apath-if-in-verts: ⟦graph G; connected G; v ∈ verts G⟧ ⟹ ∃ p. G.apath
root p v
  using G.reachable-apath by (simp add: graph.connected-iff-reachable root-in-G)

lemma undir-unique-awalk: ⟦undirected-tree G; v ∈ verts G⟧ ⟹ ∃!p. G.apath root
p v
  using undir-path-in-dir apath-if-in-verts awalk-G-T-root Suc-card-if-graph
  by (metis G.awalkI-apath unique-awalk-All undirected-tree.axioms(1) undirected-tree.connected)

lemma apath-in-dir-if-apath-G:
  assumes undirected-tree G G.apath root p v
  shows apath root p v
  using undir-path-in-dir[OF assms] assms(2) G.awalkI-apath apath-if-awalk awalk-G-T-root
by force

end

locale bfs-locale =
  fixes bfs :: ('a, 'b) pre-digraph ⇒ 'a ⇒ ('a, 'b) pre-digraph
  assumes bfs-correct: ⟦wf-digraph G; r ∈ verts G; bfs G r = T⟧ ⟹ bfs-tree G T
r

locale undir-tree-todir = undirected-tree G + bfs-locale bfs
  for G :: ('a, 'b) pre-digraph
  and bfs :: ('a, 'b) pre-digraph ⇒ 'a ⇒ ('a, 'b) pre-digraph
begin

abbreviation dir-tree-r :: 'a ⇒ ('a, 'b) pre-digraph where
  dir-tree-r ≡ bfs G

lemma directed-tree-r: r ∈ verts G ⟹ directed-tree (dir-tree-r r) r
  using bfs-correct bfs-tree.axioms(1) wf-digraph-axioms by fast

lemma bfs-dir-tree-r: r ∈ verts G ⟹ bfs-tree G (dir-tree-r r) r
  using bfs-correct wf-digraph-axioms by blast
```

**lemma** *dir-tree-r-dom-in-G*: $r \in verts\ G \Longrightarrow u \rightarrow_{dir\text{-}tree\text{-}r\ r} v \Longrightarrow u \rightarrow_G v$
**using** *bfs-dir-tree-r bfs-tree.dom-in-G* **by** *fast*

**lemma** *verts-nempty*: $verts\ G \neq \{\}$
**using** *connected connected-iff-reachable* **by** *auto*

**lemma** *card-gt0*: $card\ (verts\ G) > 0$
**using** *verts-nempty* **by** *auto*

**lemma** *Suc-card-1-eq-card*[*intro*]: $Suc\ (card\ (verts\ G) - 1) = card\ (verts\ G)$
**using** *card-gt0* **by** *simp*

**lemma** *verts-dir-tree-r-eq*[*simp*]: $r \in verts\ G \Longrightarrow verts\ (dir\text{-}tree\text{-}r\ r) = verts\ G$
**using** *bfs-tree.connected-verts-G-eq-T*[*OF bfs-dir-tree-r graph-axioms connected*]
**by** *blast*

**lemma** *tail-dir-tree-r-eq*: $r \in verts\ G \Longrightarrow tail\ (dir\text{-}tree\text{-}r\ r)\ e = tail\ G\ e$
**using** *bfs-tree.tailT-eq-tailG*[*OF bfs-dir-tree-r*] **by** *simp*

**lemma** *head-dir-tree-r-eq*: $r \in verts\ G \Longrightarrow head\ (dir\text{-}tree\text{-}r\ r)\ e = head\ G\ e$
**using** *bfs-tree.headT-eq-headG*[*OF bfs-dir-tree-r*] **by** *simp*

**lemma** *awalk-verts-G-T*: $r \in verts\ G \Longrightarrow awalk\text{-}verts = pre\text{-}digraph.awalk\text{-}verts$
$(dir\text{-}tree\text{-}r\ r)$
**using** *bfs-tree.awalk-verts-G-T bfs-dir-tree-r* **by** *fastforce*

**lemma** *dir-tree-r-all-reach*: $[\![r \in verts\ G; v \in verts\ G]\!] \Longrightarrow r \rightarrow^*_{dir\text{-}tree\text{-}r\ r} v$
**using** *directed-tree.reachable-from-root directed-tree-r verts-dir-tree-r-eq* **by** *fast*

**lemma** *fin-verts-dir-tree-r-eq*: $r \in verts\ G \Longrightarrow finite\ (verts\ (dir\text{-}tree\text{-}r\ r))$
**using** *verts-dir-tree-r-eq* **by** *auto*

**lemma** *fin-arcs-dir-tree-r-eq*: $r \in verts\ G \Longrightarrow finite\ (arcs\ (dir\text{-}tree\text{-}r\ r))$
**using** *fin-verts-dir-tree-r-eq directed-tree.verts-finite-imp-arcs-finite directed-tree-r*
**by** *fast*

**lemma** *fin-directed-tree-r*: $r \in verts\ G \Longrightarrow finite\text{-}directed\text{-}tree\ (dir\text{-}tree\text{-}r\ r)\ r$
**unfolding** *finite-directed-tree-def fin-digraph-def fin-digraph-axioms-def*
**using** *directed-tree.axioms*(*1*) *directed-tree-r fin-arcs-dir-tree-r-eq verts-dir-tree-r-eq*
**by** *force*

**lemma** *arcs-eq-2verts*: $card\ (arcs\ G) = 2 * (card\ (verts\ G) - 1)$
**using** *bfs-tree.arcs-undir-G-eq-2vertsG*[*OF bfs-dir-tree-r undirected-tree-axioms*]
*card-gt0*
**by** *fastforce*

**lemma** *arcs-compl-un-eq-arcs*:
$r \in verts\ G \Longrightarrow$
$\{e2 \in arcs\ G.\ \exists\ e1 \in arcs\ (dir\text{-}tree\text{-}r\ r).\ head\ G\ e2 = tail\ G\ e1 \wedge head\ G\ e1 =$

*tail G e2*}
∪ *arcs* (*dir-tree-r r*) = *arcs G*
**using** *bfs-tree.undir-arcs-compl-un-eq-arcs*[*OF bfs-dir-tree-r undirected-tree-axioms*]
**by** *blast*

**lemma** *unique-apath*: ⟦*u* ∈ *verts G*; *v* ∈ *verts G*⟧ ⟹ ∃!*p*. *apath u p v*
**using** *bfs-tree.undir-unique-awalk*[*OF bfs-dir-tree-r undirected-tree-axioms*] **by** *blast*

**lemma** *apath-in-dir-if-apath-G*: *apath r p v* ⟹ *pre-digraph.apath* (*dir-tree-r r*) *r p v*
**using** *bfs-tree.apath-in-dir-if-apath-G bfs-dir-tree-r undirected-tree-axioms awalkI-apath*
**by** *fast*

**lemma** *apath-verts-sub-awalk*:
⟦*apath u p1 v*; *awalk u p2 v*⟧ ⟹ *set* (*awalk-verts u p1*) ⊆ *set* (*awalk-verts u p2*)
**using** *unique-apath-verts-sub-awalk unique-apath* **by** *blast*

**lemma** *dir-tree-arc1-in-apath*:
**assumes** *u* →*dir-tree-r r v* **and** *r* ∈ *verts G*
**shows** ∃ *p*. *apath r p v* ∧ *u* ∈ *set* (*awalk-verts r p*)
**using** *directed-tree.apath-over-inarc-if-dominated*[*OF directed-tree-r*[*OF assms*(*2*)] *assms*(*1*)]
*bfs-tree.apath-sub-imp-apath bfs-dir-tree-r*[*OF assms*(*2*)] *bfs-tree.awalk-verts-G-T*
**by** *fastforce*

**lemma** *dir-tree-arc1-in-awalk*:
⟦*u* →*dir-tree-r r v*; *r* ∈ *verts G*; *awalk r p v*⟧ ⟹ *u* ∈ *set* (*awalk-verts r p*)
**using** *dir-tree-arc1-in-apath apath-verts-sub-awalk* **by** *blast*

**end**

### 6.1.2 Tranformation using PSP-Trees

Assumes existence of a conversion function that contains the n nearest nodes. This sections proves that such a generated tree contains all vertices in a connected graph.

**locale** *find-psp-tree-locale* =
**fixes** *find-psp-tree* :: ('*a*, '*b*) *pre-digraph* ⇒ ('*b* ⇒ *real*) ⇒ '*a* ⇒ *nat* ⇒ ('*a*, '*b*) *pre-digraph*
**assumes** *find-psp-tree*: ⟦*r* ∈ *verts G*; *find-psp-tree G w r n* = *T*⟧ ⟹ *psp-tree G T w r n*

**context** *psp-tree*
**begin**

**lemma** *dom-in-G*: *u* →*T v* ⟹ *u* →*G v*
**by** (*simp add*: *G.adj-mono sub-G*)

**lemma** *tailT-eq-tailG*: *tail T = tail G*
**using** *sub-G* **by** (*simp add*: *Digraph-Component.subgraph-def compatible-def*)

**lemma** *headT-eq-headG*: *head T = head G*
**using** *sub-G* **by** (*simp add*: *Digraph-Component.subgraph-def compatible-def*)

**lemma** *verts-T-subset-G*: *verts T ⊆ verts G*
**by** (*metis awalk-sub-imp-awalk G.awalk-last-in-verts subsetI unique-awalk*)

**lemma** *reachable-verts-G-subset-T*:
**assumes** *fin-digraph G*
**and** *∀ x∈verts G. source →*G x*
**and** *Suc n = card* (*verts G*)
**shows** *verts T ⊇ verts G*
**proof**(*cases card* (*verts G*))
**case** *0*
**have** *finite* (*verts G*) **using** *fin-digraph.finite-verts graph-def assms*(*1*) **by** *blast*
**then show** *?thesis* **using** *assms*(*3*) *0* **by** *simp*
**next**
**case** (*Suc n*)
**then have** *r-in-G*: *source ∈ verts G* **using** *source-in-G assms* **by** *blast*
**show** *?thesis*
**proof**(*cases n=0*)
**case** *True*
**then have** *card* (*verts G*) = *1* **using** *assms*(*3*) *Suc* **by** *auto*
**then have** *verts G* = {*source*} **using** *mem-card1-singleton r-in-G* **by** *fast*
**then show** *?thesis*
**using** *ex-sp-eq-dia in-sccs-verts-conv-reachable insert-not-empty G.reachable-in-verts*(*1*)
**by** (*metis G.reachable-mono non-empty reachable-refl sccs-verts-subsets singleton-iff sub-G*)
**next**
**case** *False*
**then obtain** *n′* **where** *n′-def*[*simp*]: *n′ = n − 1 ∧ n ≠ n′* **by** *simp*
**show** *?thesis*
**proof**(*rule ccontr*)
**assume** *¬*(*verts T ⊇ verts G*)
**then have** *strict-sub*: *verts T ⊂ verts G* **using** *psp-tree-axioms verts-T-subset-G*
**by** *fast*
**then obtain** *x* **where** *x-def*: *x ∉ verts T ∧ x ∈ verts G* **by** *blast*
**then have** *x-reach*: *source →*G x* **using** *assms*(*2*) **by** *simp*
**have** *finite* (*verts G*) **using** *fin-digraph.finite-verts graph-def assms*(*1*) **by**
*blast*
**with** *strict-sub* **have** *T-lt-G*: *card* (*verts T*) *<* *card* (*verts G*) **by** (*simp add*:
*psubset-card-mono*)
**then have** *T-le-n*: *card* (*verts T*) *≤ n* **using** *Suc assms*(*3*) **by** *simp*
**have** *G.n-nearest-verts w source n* (*verts T*)
**using** *Suc assms*(*3*) *partial* **by** *simp*
**then have** *1*: *G.n-nearest-verts w source* (*Suc n′*) (*verts T*) **using** *n′-def* **by**
*simp*

**then obtain** *U* **where** *U-def*[*simp*]: $U \subseteq verts\ T \wedge G.n\text{-}nearest\text{-}verts\ w\ source\ n'\ U$
**using** *Zero-not-Suc diff-Suc-1 equalityE G.nnvs-ind-cases subset-insertI* **by** *metis*
**then show** *False*
**proof**(*cases G.unvisited-verts source U* $\neq \{\}$)
**case** *True*
**then have** *card U* $\geq$ *Suc n'* **using** *U-def fin-digraph.nnvs-card-ge-n assms*(*1*) **by** *fast*
**then have** *U-Suc-n'*: *card U* = *Suc n'* **using** *1 U-def G.nnvs-card-le-n* **by** *force*
**have** *G.nearest-vert w source U* $\in$ *G.unvisited-verts source U*
**using** *True assms*(*1*) **by** (*simp add*: *fin-digraph.nearest-vert-unvis*)
**then have** *G.nearest-vert w source U* $\notin$ *U* **using** *G.unvisited-verts-def* **by** *simp*
**then have** *U-ins-Suc2-n'*: *card* (*insert* (*G.nearest-vert w source U*) *U*) = *Suc* (*Suc n'*)
**using** *U-Suc-n' card-Suc-eq* **by** *blast*
**have** *card* (*verts T*) $\leq$ *Suc n'* **using** *T-le-n* **by** *simp*
**moreover have** *card U* $\leq$ *card* (*verts T*) **by** (*simp add*: *card-mono*)
**ultimately have** *T-Suc-n'*: *card* (*verts T*) = *Suc n'* **using** *U-Suc-n'* **by** *simp*
**then have** *U-eq-T*: *U* = *verts T* **by** (*simp add*: *U-Suc-n' card-seteq*)
**have** *card* (*insert* (*G.nearest-vert w source U*) *U*) = *card* (*verts T*)
**using** *True U-eq-T U-ins-Suc2-n' 1* **by** (*metis fin-digraph.nnvs-card-eq-n assms*(*1*))
**then show** *?thesis* **using** *T-Suc-n' U-ins-Suc2-n'* **by** *linarith*
**next**
**case** *False*
**have** *x* $\notin$ *U* **using** *x-def U-def* **by** *blast*
**then have** *G.unvisited-verts source U* $\neq \{\}$
**using** *G.unvisited-verts-def x-def x-reach* **by** *blast*
**then show** *?thesis* **using** *False* **by** *simp*
**qed**
**qed**
**qed**
**qed**

**lemma** *reachable-verts-G-eq-T*:
⟦*fin-digraph G*; $\forall x{\in}verts\ G.\ source \rightarrow^*_G x$; *Suc n* = *card* (*verts G*)⟧ $\Longrightarrow$ *verts T* = *verts G*
**by** (*simp add*: *reachable-verts-G-subset-T set-eq-subset verts-T-subset-G*)

**lemma** *connected-verts-G-eq-T*:
**assumes** *graph G*
**and** *connected G*
**and** *Suc n* = *card* (*verts G*)
**shows** *verts T* = *verts G*
**proof** −

**have** *0*: *fin-digraph G* **using** *assms*(*1*) *graph.axioms*(*1*) *digraph.axioms*(*1*) **by** *blast*
**have** *source* ∈ *verts G* **using** *source-in-G* **by** *fast*
**then have** ∀ *x*∈*verts G*. *source* →**G x* **using** *graph.connected-iff-reachable assms*(*1*,*2*) **by** *blast*
**then show** *?thesis* **using** *assms*(*3*) *reachable-verts-G-eq-T 0* **by** *blast*
**qed**

**lemma** *con-Suc-card-arcs-eq-card-verts*:
**assumes** *graph G*
**and** *connected G*
**and** *Suc n* = *card* (*verts G*)
**shows** *Suc* (*card* (*arcs T*)) = *card* (*verts G*)
**using** *Suc-card-arcs-eq-card-verts connected-verts-G-eq-T assms* **by** *fastforce*

**lemma** *reverse-arc-in-G*:
**assumes** *graph G* **and** *e1* ∈ *arcs T*
**shows** ∃ *e2* ∈ *arcs G*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*
**proof** −
**interpret** *graph G* **using** *assms*(*1*) **.**
**have** *e1* ∈ *arcs G* **using** *assms*(*2*) *sub-G* **by** *blast*
**then show** *?thesis* **using** *sym-arcs symmetric-conv* **by** *fastforce*
**qed**

**lemma** *reverse-arc-notin-T*:
**assumes** *e1* ∈ *arcs T* **and** *head G e2* = *tail G e1* **and** *head G e1* = *tail G e2*
**shows** *e2* ∉ *arcs T*
**proof**
**assume** *asm*: *e2* ∈ *arcs T*
**then have** *tail T e2* →*T head T e2* **by** (*simp add*: *in-arcs-imp-in-arcs-ends*)
**then have** *head G e1* →*T tail G e1*
**using** *assms*(*2*,*3*) *sub-G* **by**(*simp add*: *Digraph-Component.subgraph-def compatible-def*)
**moreover have** *tail G e1* →*T head G e1*
**using** *assms*(*1*) *sub-G*
**by**(*simp add*: *Digraph-Component.subgraph-def compatible-def in-arcs-imp-in-arcs-ends*)
**ultimately show** *False* **using** *reachable1-not-reverse* **by** *blast*
**qed**

**lemma** *reverse-arc-in-G-only*:
**assumes** *graph G* **and** *e1* ∈ *arcs T*
**shows** ∃ *e2* ∈ *arcs G*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2* ∧ *e2* ∉ *arcs T*
**using** *reverse-arc-in-G reverse-arc-notin-T assms* **by** *blast*

**lemma** *no-multi-T-G*:
**assumes** *e1* ∈ *arcs T* **and** *e2* ∈ *arcs T* **and** *e1* ≠ *e2*
**shows** *head G e1* ≠ *head G e2* ∨ *tail G e1* ≠ *tail G e2*
**using** *nomulti.no-multi-arcs assms sub-G*

**by**(*auto simp*: *Digraph-Component.subgraph-def compatible-def arc-to-ends-def*)

**lemma** *T-arcs-compl-fin*:
**assumes** *fin-digraph G* **and** *es* ⊆ *arcs T*
**shows** *finite* {*e2*∈ *arcs G*. (∃ *e1* ∈ *es*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*)}
**using** *assms fin-digraph.finite-arcs* **by** *fastforce*

**corollary** *T-arcs-compl-fin'*:
**assumes** *graph G* **and** *es* ⊆ *arcs T*
**shows** *finite* {*e2*∈ *arcs G*. (∃ *e1* ∈ *es*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*)}
**using** *assms T-arcs-compl-fin graph.axioms*(*1*) *digraph.axioms*(*1*) **by** *blast*

**lemma** *T-arcs-compl-card-eq*:
**assumes** *graph G* **and** *es* ⊆ *arcs T*
**shows** *card* {*e2*∈ *arcs G*. (∃ *e1* ∈ *es*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*)} = *card es*
**using** *finite-subset*[*OF assms*(*2*) *finite-arcs*] *assms* **proof**(*induction es rule*: *finite-induct*)
**case** (*insert e1 es*)
**let** *?ees* = {*e2* ∈ *arcs G*. ∃ *e1*∈*insert e1 es*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*}
**let** *?es* = {*e2* ∈ *arcs G*. ∃ *e1*∈*es*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*}
**obtain** *e2* **where** *e2-def*: *e2* ∈ *arcs G head G e2* = *tail G e1 head G e1* = *tail G e2*
**using** *reverse-arc-in-G-only insert.prems* **by** *blast*
**then have** *e2-notin*: *e2* ∉ {*e2* ∈ *arcs G*. ∃ *e1*∈*es*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*}
**using** *insert.hyps*(*2*) *insert.prems*(*2*) *no-multi-T-G* **by** *fastforce*
**have** ∀ *e3* ∈ *arcs G*. *e2* = *e3* ∨ *head G e3* ≠ *head G e2* ∨ *tail G e3* ≠ *tail G e2*
**using** *e2-def*(*1*) *nomulti-digraph.no-multi-alt digraph.axioms*(*3*) *graph.axioms*(*1*) *insert.prems*(*1*)
**by** *fast*
**then have** *?ees* = *insert e2 ?es* **using** *e2-def* **by** *auto*
**moreover have** *finite ?es* **using** *insert.prems T-arcs-compl-fin'* **by** *simp*
**ultimately have** *card ?ees* = *Suc* (*card ?es*) **using** *e2-notin* **by** *simp*
**then show** *?case* **using** *insert* **by** *force*
**qed**(*simp*)

**lemma** *arcs-graph-G-ge-2vertsT*:
**assumes** *graph G*
**shows** *card* (*arcs G*) ≥ *2* ∗ (*card* (*verts T*) − *1*)
**proof** −
**let** *?compl* = {*e2*∈ *arcs G*. (∃ *e1* ∈ *arcs T*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*)}
**interpret** *graph G* **by** (*rule assms*)
**have** ∀ *e1* ∈ *arcs T*. ∃ *e2* ∈ *arcs G*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail*

*G e2*
**using** *reverse-arc-in-G-only assms* **by** *blast*
**have** *?compl ∩ arcs T = {}* **using** *reverse-arc-notin-T* **by** *blast*
**then have** *card (?compl ∪ arcs T) = card ?compl + card (arcs T)* **by** (*simp add*: *card-Un-disjoint*)
**moreover have** *?compl ∪ arcs T ⊆ arcs G* **using** *sub-G* **by** *blast*
**moreover have** *finite (arcs G)* **by** *simp*
**ultimately have** *card ?compl + card (arcs T) ≤ card (arcs G)*
**using** *card-mono*[*of arcs G ?compl ∪ arcs T*] **by** *presburger*
**moreover have** *card (arcs T) = (card (verts T) − 1)*
**using** *Suc-card-arcs-eq-card-verts assms* **by** *fastforce*
**ultimately show** *?thesis* **using** *T-arcs-compl-card-eq* **by** *fastforce*
**qed**

**lemma** *arcs-graph-G-ge-2vertsG*:
⟦*graph G*; *connected G*; *Suc n = card (verts G)*⟧ ⟹ *card (arcs G) ≥ 2 ∗ (card (verts G) − 1)*
**using** *arcs-graph-G-ge-2vertsT connected-verts-G-eq-T* **by** *simp*

**lemma** *arcs-undir-G-eq-2vertsG*:
⟦*undirected-tree G*; *Suc n = card (verts G)*⟧ ⟹ *card (arcs G) = 2 ∗ (card (verts G) − 1)*
**using** *arcs-graph-G-ge-2vertsG undirected-tree.acyclic undirected-tree.axioms*(*1*) *undirected-tree.connected* **by** *fastforce*

**lemma** *undir-arcs-compl-un-eq-arcs*:
**assumes** *undirected-tree G* **and** *Suc n = card (verts G)*
**shows** {*e2*∈ *arcs G*. (∃ *e1* ∈ *arcs T*. *head G e2 = tail G e1* ∧ *head G e1 = tail G e2*)} ∪ *arcs T*
= *arcs G*
**proof** −
**let** *?compl* = {*e2*∈ *arcs G*. (∃ *e1* ∈ *arcs T*. *head G e2 = tail G e1* ∧ *head G e1 = tail G e2*)}
**interpret** *undirected-tree G* **using** *assms*(*1*) *undirected-tree.axioms*(*1*) **by** *fast*
**have** *?compl ∩ arcs T = {}* **using** *reverse-arc-notin-T* **by** *blast*
**then have** *0*: *card (?compl ∪ arcs T) = card ?compl + card (arcs T)*
**by** (*simp add*: *card-Un-disjoint*)
**have** *card (arcs T) = (card (verts T) − 1)* **using** *Suc-card-arcs-eq-card-verts assms* **by** *fastforce*
**then have** *card ?compl + card (arcs T) = 2 ∗ (card (verts G) − 1)*
**using** *T-arcs-compl-card-eq connected-verts-G-eq-T connected assms*(*2*) **by** *fastforce*
**moreover have** *card (arcs G) = 2 ∗ (card (verts G) − 1)*
**using** *assms arcs-undir-G-eq-2vertsG* **by** *blast*
**moreover have** *?compl ∪ arcs T ⊆ arcs G* **using** *sub-G* **by** *blast*
**ultimately show** *?thesis* **by** (*simp add*: *0 card-subset-eq*)
**qed**

**lemma** *split-fst-nonelem*:

⟦¬*set xs* ⊆ *X*; *set xs* ⊆ *Y*⟧ ⟹ ∃ *x ys zs*. *ys*@*x*#*zs*=*xs* ∧ *x* ∉ *X* ∧ *x* ∈ *Y* ∧ *set ys* ⊆ *X*
**proof**(*induction xs*)
**case** (*Cons x xs*)
**then show** *?case*
**proof**(*cases x* ∈ *X*)
**case** *True*
**then obtain** *z ys zs* **where** *ys-def*: *ys*@*z*#*zs*=*xs z* ∉ *X z* ∈ *Y set ys* ⊆ *X* **using** *Cons* **by** *auto*
**then have** *set* (*x*#*ys*) ⊆ *X* **using** *True* **by** *simp*
**then show** *?thesis* **using** *ys-def*(*1*−*3*) *append-Cons* **by** *fast*
**next**
**case** *False*
**then show** *?thesis* **using** *Cons.prems*(*2*) **by** *fastforce*
**qed**
**qed**(*simp*)

**lemma** *source-no-inarc-T*: *head G e* = *source* ⟹ *e* ∉ *arcs T*
**using** *in-arcs-root sub-G* **by** (*auto simp*: *Digraph-Component.subgraph-def compatible-def*)

**lemma** *source-all-outarcs-T*:
⟦*undirected-tree G*; *Suc n* = *card* (*verts G*); *tail G e* = *source*; *e* ∈ *arcs G*⟧ ⟹ *e* ∈ *arcs T*
**using** *source-no-inarc-T undir-arcs-compl-un-eq-arcs* **by** *blast*

**lemma** *cas-G-T*: *G.cas* = *cas*
**using** *sub-G compatible-cas* **by** *fastforce*

**lemma** *awalk-G-T*: *u* ∈ *verts T* ⟹ *set p* ⊆ *arcs T* ⟹ *G.awalk u p* = *awalk u p*
**using** *cas-G-T awalk-def G.awalk-def sub-G* **by** *fastforce*

**corollary** *awalk-G-T-root*: *set p* ⊆ *arcs T* ⟹ *G.awalk source p* = *awalk source p*
**using** *awalk-G-T root-in-T* **by** *blast*

**lemma** *awalk-verts-G-T*: *G.awalk-verts* = *awalk-verts*
**using** *sub-G compatible-awalk-verts* **by** *blast*

**lemma** *apath-sub-imp-apath*: *apath u p v* ⟹ *G.apath u p v*
**by** (*simp add*: *G.apath-def apath-def awalk-sub-imp-awalk awalk-verts-G-T*)

**lemma** *outarc-inT-if-head-not-inarc*:
**assumes** *undirected-tree G* **and** *Suc n* = *card* (*verts G*)
**and** *tail G e2* = *v* **and** *e2* ∈ *arcs G* **and** *head G e2* ≠ *u* **and** *u* →$_T$ *v*
**shows** *e2* ∈ *arcs T*
**proof** (*rule ccontr*)
**let** *?compl* = {*e2*∈ *arcs G*. (∃ *e1* ∈ *arcs T*. *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*)}
**assume** *e2* ∉ *arcs T*

**then have** *e2 ∈ ?compl* **using** *assms(4) undir-arcs-compl-un-eq-arcs*[*OF assms*(*1−2*)]
**by** *blast*
**then obtain** *e1* **where** *e1-def*: *e1 ∈ arcs T head G e2 = tail T e1 head T e1 = v*
**using** *sub-G assms*(*3*) **by** (*auto simp*: *Digraph-Component.subgraph-def compatible-def*)
**obtain** *e* **where** *e ∈ arcs T tail T e = u head T e = v* **using** *assms*(*6*) **by** *blast*
**then show** *False* **using** *two-in-arcs-contr e1-def assms*(*5*) **by** *blast*
**qed**

**corollary** *reverse-arc-if-out-arc-undir*:
⟦*undirected-tree G*; *Suc n = card* (*verts G*); *tail G e2 = v*; *e2 ∈ arcs G*; *e2 ∉ arcs T*; *u* →$_T$ *v*⟧
⟹ *head G e2 = u*
**using** *outarc-inT-if-head-not-inarc* **by** *blast*

**lemma** *undir-path-in-dir*:
**assumes** *undirected-tree G Suc n = card* (*verts G*) *G.apath source p v*
**shows** *set p ⊆ arcs T*
**proof** (*rule ccontr*)
**assume** *asm*: ¬*set p ⊆ arcs T*
**have** *set p ⊆ arcs G* **using** *assms*(*3*) *G.apath-def G.awalk-def* **by** *fast*
**then obtain** *e p1 p2* **where** *e-def*: *p1 @ e # p2 = p e ∉ arcs T e ∈ arcs G set p1 ⊆ arcs T*
**using** *split-fst-nonelem*[*OF asm*, *of arcs G*] **by** *auto*
**show** *False*
**proof**(*cases p1=[]*)
**case** *True*
**then have** *tail G e = source* **using** *assms*(*3*) *e-def*(*1*) *G.apath-Cons-iff* **by** *auto*
**then show** *?thesis* **using** *source-all-outarcs-T*[*OF assms*(*1−2*)] *e-def*(*2*,*3*) **by** *blast*
**next**
**case** *False*
**then have** *awalk-G*: *G.awalk source* (*p1 @ e # p2*) *v*
**using** *assms*(*3*) *pre-digraph.apath-def e-def*(*1*) **by** *fast*
**then have** *G.awalk source p1* (*tail G e*) **by** *force*
**then have** *awalk-p1T*: *awalk source p1* (*tail T e*)
**using** *e-def*(*4*) *sub-G cas-G-T root-in-T*
**by** (*simp add*: *Digraph-Component.subgraph-def pre-digraph.awalk-def compatible-def*)
**then have** *source* →$^+_T$ *tail T e* **using** *False reachable1-awalkI* **by** *auto*
**then obtain** *u* **where** *u-def*: *u* →$_T$ *tail T e* **using** *tranclD2* **by** *metis*
**have** *tail T e = tail G e*
**using** *sub-G* **by** (*simp add*: *Digraph-Component.subgraph-def compatible-def*)
**then have** *hd-e-u*: *head G e = u*
**using** *reverse-arc-if-out-arc-undir*[*OF assms*(*1−2*)] *u-def e-def*(*2*,*3*) **by** *simp*
**have** *head T* (*last p1*) = *tail T e* **using** *False awalk-p1T awalk-verts-conv* **by** *fastforce*

**then have** *tail T (last p1) = u*
**using** *False u-def e-def(4) two-in-arcs-contr last-in-set* **by** *fastforce*
**then have** *0*: *tail G (last p1) = u*
**using** *sub-G* **by** (*simp add*: *Digraph-Component.subgraph-def compatible-def*)
**obtain** *ps* **where** *ps* @ [*last p1*] = *p1* **using** *False append-butlast-last-id* **by** *auto*
**then have** *ps-def*: *ps* @ [*last p1*] @ *e* # *p2* = *p* **using** *e-def* **by** *auto*
**then have** *awalk-G*: *G.awalk source* (*ps* @ [*last p1*] @ *e* # *p2*) *v*
**using** *assms(3)* **by** (*simp add*: *pre-digraph.apath-def*)
**have** ¬(*distinct* (*G.awalk-verts source p*))
**using** *G.not-distinct-if-head-eq-tail*[*OF 0 hd-e-u awalk-G*] *ps-def* **by** *simp*
**then show** *?thesis* **using** *assms(3) G.apath-def* **by** *blast*
**qed**
**qed**

**lemma** *source-reach-all*: ⟦*graph G*; *connected G*; *v* ∈ *verts G*⟧ ⟹ *source* →** G v*
**by** (*simp add*: *graph.connected-iff-reachable source-in-G*)

**lemma** *apath-if-in-verts*: ⟦*graph G*; *connected G*; *v* ∈ *verts G*⟧ ⟹ ∃ *p*. *G.apath source p v*
**using** *G.reachable-apath* **by** (*simp add*: *graph.connected-iff-reachable source-in-G*)

**lemma** *undir-unique-awalk*:
⟦*undirected-tree G*; *Suc n* = *card* (*verts G*); *v* ∈ *verts G*⟧ ⟹ ∃!*p*. *G.apath source p v*
**using** *undir-path-in-dir apath-if-in-verts awalk-G-T-root*
**by** (*metis G.awalkI-apath unique-awalk-All undirected-tree.axioms(1) undirected-tree.connected*)

**lemma** *apath-in-dir-if-apath-G*:
**assumes** *undirected-tree G Suc n* = *card* (*verts G*) *G.apath source p v*
**shows** *apath source p v*
**using** *undir-path-in-dir*[*OF assms*] *assms(3) G.awalkI-apath apath-if-awalk awalk-G-T-root*
**by** *force*

**end**

**locale** *undir-tree-todir-psp* = *undirected-tree G* + *find-psp-tree-locale to-psp*
**for** *G* :: ('*a*, '*b*) *pre-digraph*
**and** *to-psp* :: ('*a*, '*b*) *pre-digraph* ⇒ ('*b* ⇒ *real*) ⇒ '*a* ⇒ *nat* ⇒ ('*a*, '*b*) *pre-digraph*
**begin**

**abbreviation** *dir-tree-r* :: '*a* ⇒ ('*a*, '*b*) *pre-digraph* **where**
*dir-tree-r r* ≡ *to-psp G* (λ-. *1*) *r* (*Finite-Set.card* (*verts G*) − *1*)

**lemma** *directed-tree-r*: *r* ∈ *verts G* ⟹ *directed-tree* (*dir-tree-r r*) *r*
**using** *find-psp-tree psp-tree.axioms(1)* **by** *fast*

**lemma** *psp-dir-tree-r*:
*r* ∈ *verts G* ⟹ *psp-tree G* (*dir-tree-r r*) (λ-. *1*) *r* (*Finite-Set.card* (*verts G*) −

*1*)
**using** *find-psp-tree* **by** *blast*

**lemma** *dir-tree-r-dom-in-G*: $r \in verts\ G \Longrightarrow u \rightarrow_{dir\text{-}tree\text{-}r\ r} v \Longrightarrow u \rightarrow_G v$
**using** *psp-tree.dom-in-G psp-dir-tree-r* **by** *fast*

**lemma** *verts-nempty*: $verts\ G \neq \{\}$
**using** *connected connected-iff-reachable* **by** *auto*

**lemma** *card-gt0*: $card\ (verts\ G) > 0$
**using** *verts-nempty* **by** *auto*

**lemma** *Suc-card-1-eq-card*[*intro*]: $Suc\ (card\ (verts\ G) - 1) = card\ (verts\ G)$
**using** *card-gt0* **by** *simp*

**lemma** *verts-dir-tree-r-eq*[*simp*]: $r \in verts\ G \Longrightarrow verts\ (dir\text{-}tree\text{-}r\ r) = verts\ G$
**using** *psp-tree.connected-verts-G-eq-T*[*OF psp-dir-tree-r graph-axioms connected*]
**by** *blast*

**lemma** *tail-dir-tree-r-eq*: $r \in verts\ G \Longrightarrow tail\ (dir\text{-}tree\text{-}r\ r)\ e = tail\ G\ e$
**using** *psp-tree.tailT-eq-tailG*[*OF psp-dir-tree-r*] **by** *simp*

**lemma** *head-dir-tree-r-eq*: $r \in verts\ G \Longrightarrow head\ (dir\text{-}tree\text{-}r\ r)\ e = head\ G\ e$
**using** *psp-tree.headT-eq-headG*[*OF psp-dir-tree-r*] **by** *simp*

**lemma** *awalk-verts-G-T*: $r \in verts\ G \Longrightarrow awalk\text{-}verts = pre\text{-}digraph.awalk\text{-}verts$
$(dir\text{-}tree\text{-}r\ r)$
**using** *psp-tree.awalk-verts-G-T psp-dir-tree-r* **by** *fastforce*

**lemma** *dir-tree-r-all-reach*: $[\![r \in verts\ G; v \in verts\ G]\!] \Longrightarrow r \rightarrow^*_{dir\text{-}tree\text{-}r\ r} v$
**using** *directed-tree.reachable-from-root directed-tree-r verts-dir-tree-r-eq* **by** *fast*

**lemma** *fin-verts-dir-tree-r-eq*: $r \in verts\ G \Longrightarrow finite\ (verts\ (dir\text{-}tree\text{-}r\ r))$
**using** *verts-dir-tree-r-eq* **by** *auto*

**lemma** *fin-arcs-dir-tree-r-eq*: $r \in verts\ G \Longrightarrow finite\ (arcs\ (dir\text{-}tree\text{-}r\ r))$
**using** *fin-verts-dir-tree-r-eq directed-tree.verts-finite-imp-arcs-finite directed-tree-r*
**by** *fast*

**lemma** *fin-directed-tree-r*: $r \in verts\ G \Longrightarrow finite\text{-}directed\text{-}tree\ (dir\text{-}tree\text{-}r\ r)\ r$
**unfolding** *finite-directed-tree-def fin-digraph-def fin-digraph-axioms-def*
**using** *directed-tree.axioms*(*1*) *directed-tree-r fin-arcs-dir-tree-r-eq verts-dir-tree-r-eq*
**by** *force*

**lemma** *arcs-eq-2verts*: $card\ (arcs\ G) = 2 * (card\ (verts\ G) - 1)$
**using** *psp-tree.arcs-undir-G-eq-2vertsG*[*OF psp-dir-tree-r undirected-tree-axioms*]
*card-gt0*
**by** *fastforce*

**lemma** *arcs-compl-un-eq-arcs*:
$r \in verts\ G \Longrightarrow$
$\{e2 \in arcs\ G.\ \exists\, e1 \in arcs\ (dir\text{-}tree\text{-}r\ r).\ head\ G\ e2 = tail\ G\ e1 \wedge head\ G\ e1 = tail\ G\ e2\}$
$\cup\ arcs\ (dir\text{-}tree\text{-}r\ r) = arcs\ G$
**using** *psp-tree.undir-arcs-compl-un-eq-arcs*[*OF psp-dir-tree-r undirected-tree-axioms*]
**by** *blast*

**lemma** *unique-apath*: $[\![u \in verts\ G;\ v \in verts\ G]\!] \Longrightarrow \exists!p.\ apath\ u\ p\ v$
**using** *psp-tree.undir-unique-awalk*[*OF psp-dir-tree-r undirected-tree-axioms*] **by** *blast*

**lemma** *apath-in-dir-if-apath-G*: $apath\ r\ p\ v \Longrightarrow pre\text{-}digraph.apath\ (dir\text{-}tree\text{-}r\ r)\ r\ p\ v$
**using** *psp-tree.apath-in-dir-if-apath-G psp-dir-tree-r undirected-tree-axioms awalkI-apath*
**by** *fast*

**lemma** *apath-verts-sub-awalk*:
$[\![apath\ u\ p1\ v;\ awalk\ u\ p2\ v]\!] \Longrightarrow set\ (awalk\text{-}verts\ u\ p1) \subseteq set\ (awalk\text{-}verts\ u\ p2)$
**using** *unique-apath-verts-sub-awalk unique-apath* **by** *blast*

**lemma** *dir-tree-arc1-in-apath*:
**assumes** $u \rightarrow_{dir\text{-}tree\text{-}r\ r} v$ **and** $r \in verts\ G$
**shows** $\exists\, p.\ apath\ r\ p\ v \wedge u \in set\ (awalk\text{-}verts\ r\ p)$
**using** *directed-tree.apath-over-inarc-if-dominated*[*OF directed-tree-r*[*OF assms*(*2*)] *assms*(*1*)]
*psp-tree.apath-sub-imp-apath psp-dir-tree-r*[*OF assms*(*2*)] *psp-tree.awalk-verts-G-T*
**by** *fastforce*

**lemma** *dir-tree-arc1-in-awalk*:
$[\![u \rightarrow_{dir\text{-}tree\text{-}r\ r} v;\ r \in verts\ G;\ awalk\ r\ p\ v]\!] \Longrightarrow u \in set\ (awalk\text{-}verts\ r\ p)$
**using** *dir-tree-arc1-in-apath apath-verts-sub-awalk* **by** *blast*

**end**

## 6.2 Additions for Induction on Directed Trees

**lemma** *fin-dir-tree-single*:
*finite-directed-tree* $(\!|verts = \{r\},\ arcs = \{\},\ tail = t,\ head = h|\!)\ r$
**by** *unfold-locales* (*fastforce simp*: *pre-digraph.cas.simps*(*1*) *pre-digraph.awalk-def*)+

**corollary** *dir-tree-single*: *directed-tree* $(\!|verts = \{r\},\ arcs = \{\},\ tail = t,\ head = h|\!)\ r$
**by** (*simp add*: *fin-dir-tree-single finite-directed-tree.axioms*(*1*))

**lemma** *split-list-not-last*: $[\![y \in set\ xs;\ y \neq last\ xs]\!] \Longrightarrow \exists\, as\ bs.\ as\ @\ y\ \#\ bs = xs \wedge bs \neq []$
**using** *split-list* **by** *fastforce*

**lemma** *split-last-eq*: $[\![as \mathbin{@} y \# bs = xs;\ bs \neq []]\!] \Longrightarrow last\ bs = last\ xs$
**by** *auto*

**lemma** *split-list-last-sep*: $[\![y \in set\ xs;\ y \neq last\ xs]\!] \Longrightarrow \exists as\ bs.\ as \mathbin{@} y \# bs \mathbin{@} [last\ xs] = xs$
**using** *split-list-not-last*[*of y xs*] *split-last-eq append-butlast-last-id* **by** *metis*

**context** *directed-tree*
**begin**

**lemma** *root-if-all-reach*: $\forall v \in verts\ T.\ x \rightarrow^*_T v \Longrightarrow x = root$
**proof**(*rule ccontr*)
**assume** *assms*: $\forall v \in verts\ T.\ x \rightarrow^*_T v\ x \neq root$
**then have** $x \rightarrow^*_T root$ **by** (*simp add*: *root-in-T*)
**then have** $\exists x.\ x \rightarrow_T root$ **using** *assms*(*2*) **by** (*auto elim*: *trancl.cases*)
**then show** *False* **using** *dominated-not-root* **by** *blast*
**qed**

**lemma** *add-leaf-cas-preserv*:
**fixes** *u v a*
**defines** $T' \equiv (\!|verts = verts\ T \cup \{v\},\ arcs = arcs\ T \cup \{a\},$
$tail = (tail\ T)(a := u),\ head = (head\ T)(a := v)|\!)$
**assumes** $a \notin arcs\ T$ **and** $set\ p \subseteq arcs\ T$ **and** $cas\ x\ p\ y$
**shows** *pre-digraph.cas* $T'\ x\ p\ y$
**using** *assms* **proof**(*induction p arbitrary*: *x*)
**case** (*Cons p ps*)
**then have** $tail\ T'\ p = x$ **by** *auto*
**moreover have** *pre-digraph.cas* $T'$ $(head\ T'\ p)$ *ps y* **using** *Cons* **by** *force*
**ultimately show** *?case* **using** *pre-digraph.cas.simps*(*2*) **by** *fast*
**qed**(*simp add*: *pre-digraph.cas.simps*(*1*))

**lemma** *add-leaf-awalk-preserv*:
**fixes** *u v a*
**defines** $T' \equiv (\!|verts = verts\ T \cup \{v\},\ arcs = arcs\ T \cup \{a\},$
$tail = (tail\ T)(a := u),\ head = (head\ T)(a := v)|\!)$
**assumes** $a \notin arcs\ T$ **and** *awalk x p y*
**shows** *pre-digraph.awalk* $T'\ x\ p\ y$
**using** *assms add-leaf-cas-preserv* **unfolding** *pre-digraph.awalk-def* **by** *auto*

**lemma** *add-leaf-awalk-T*:
**fixes** *u v a*
**defines** $T' \equiv (\!|verts = verts\ T \cup \{v\},\ arcs = arcs\ T \cup \{a\},$
$tail = (tail\ T)(a := u),\ head = (head\ T)(a := v)|\!)$
**assumes** $a \notin arcs\ T$ **and** $x \in verts\ T$
**shows** $\exists p.$ *pre-digraph.awalk* $T'$ *root p x*
**using** *add-leaf-awalk-preserv assms unique-awalk*[*of x*] **by** *blast*

**lemma** (**in** *pre-digraph*) *cas-append-if*:
$[\![cas\ x\ ps\ u;\ tail\ G\ p = u;\ head\ G\ p = v]\!] \Longrightarrow cas\ x\ (ps@[p])\ v$

**using** *cas-append-iff*[*of x ps*] **by** (*metis append.right-neutral cas.simps*)

**lemma** *add-leaf-awalk-T-new*:
**fixes** *u v a*
**defines** $T' \equiv (\!|verts = verts\ T \cup \{v\},\ arcs = arcs\ T \cup \{a\},$
$tail = (tail\ T)(a := u),\ head = (head\ T)(a := v)|\!)$
**assumes** $a \notin arcs\ T$ **and** $u \in verts\ T$
**shows** $\exists p.$ *pre-digraph.awalk* $T'$ *root p v*
**proof** −
**obtain** *ps* **where** *ps-def*: *root* $\in$ *verts* $T'$ *set ps* $\subseteq$ *arcs* $T'$ *pre-digraph.cas* $T'$ *root ps u*
**using** *add-leaf-awalk-T assms* **unfolding** *pre-digraph.awalk-def* **by** *blast*
**have** *pre-digraph.cas* $T'$ *root* (*ps*@[*a*]) *v*
**using** *pre-digraph.cas-append-if*[*OF ps-def*(*3*)] *assms*(*1*) **by** *simp*
**moreover have** *set* (*ps*@[*a*]) $\subseteq$ *arcs* $T'$ **using** *ps-def*(*2*) *assms*(*1*) **by** *simp*
**ultimately show** *?thesis* **using** *ps-def*(*1*) **unfolding** *pre-digraph.awalk-def* **by** *blast*
**qed**

**lemma** *add-leaf-cas-orig*:
**fixes** *u v a*
**defines** $T' \equiv (\!|verts = verts\ T \cup \{v\},\ arcs = arcs\ T \cup \{a\},$
$tail = (tail\ T)(a := u),\ head = (head\ T)(a := v)|\!)$
**assumes** $a \notin arcs\ T$ **and** *set p* $\subseteq$ *arcs T* **and** *pre-digraph.cas* $T'$ *x p y*
**shows** *cas x p y*
**using** *assms* **proof**(*induction p arbitrary*: *x*)
**case** (*Cons p ps*)
**then have** *tail* $T'$ *p* = *x* **using** *pre-digraph.cas.simps*(*2*) **by** *fast*
**then have** *tail T p* = *x* **using** *Cons.prems*(*1*,*2*) *Cons.hyps*(*2*) **by** *auto*
**moreover have** *head* $T'$ *p* = *head T p* **using** *Cons.prems*(*1*,*2*) *Cons.hyps*(*2*) **by** *auto*
**moreover have** *pre-digraph.cas* $T'$ (*head* $T'$ *p*) *ps y*
**using** *Cons.prems*(*3*) *pre-digraph.cas.simps*(*2*) **by** *fast*
**ultimately show** *?case* **using** *Cons* **by** *simp*
**qed**(*simp add*: *pre-digraph.cas.simps*(*1*))

**lemma** *add-leaf-awalk-orig-aux*:
**fixes** *u v a*
**defines** $T' \equiv (\!|verts = verts\ T \cup \{v\},\ arcs = arcs\ T \cup \{a\},$
$tail = (tail\ T)(a := u),\ head = (head\ T)(a := v)|\!)$
**assumes** $a \notin arcs\ T$ **and** $x \in verts\ T$ **and** *set p* $\subseteq$ *arcs T* **and** *pre-digraph.awalk* $T'$ *x p y*
**shows** *awalk x p y*
**using** *assms add-leaf-cas-orig* **unfolding** *pre-digraph.awalk-def* **by** *blast*

**lemma** *add-leaf-cas-xT-if-yT*:
**fixes** *u v a*
**defines** $T' \equiv (\!|verts = verts\ T \cup \{v\},\ arcs = arcs\ T \cup \{a\},$
$tail = (tail\ T)(a := u),\ head = (head\ T)(a := v)|\!)$

**assumes** $u \in verts\ T$ **and** $y \in verts\ T$ **and** $set\ p \subseteq arcs\ T'$ **and** *pre-digraph.cas* $T'\ x\ p\ y$
**shows** $x \in verts\ T$
**using** *assms* **by** (*induction p arbitrary*: $x$) (*auto simp*: *pre-digraph.cas.simps*)

**lemma** *add-leaf-cas-xT-arcsT-if-yT*:
**fixes** $u\ v\ a$
**defines** $T' \equiv (\!|verts = verts\ T \cup \{v\},\ arcs = arcs\ T \cup \{a\},$
$tail = (tail\ T)(a := u),\ head = (head\ T)(a := v)|\!)$
**assumes** $v \notin verts\ T$ **and** $y \in verts\ T$ **and** $set\ p \subseteq arcs\ T'$ **and** *pre-digraph.cas* $T'\ x\ p\ y$
**shows** $set\ p \subseteq arcs\ T$ **and** $x \in verts\ T$
**using** *assms* **by** (*induction p arbitrary*: $x$) (*auto simp*: *pre-digraph.cas.simps*)

**lemma** *add-leaf-awalk-orig*:
**fixes** $u\ v\ a$
**defines** $T' \equiv (\!|verts = verts\ T \cup \{v\},\ arcs = arcs\ T \cup \{a\},$
$tail = (tail\ T)(a := u),\ head = (head\ T)(a := v)|\!)$
**assumes** $a \notin arcs\ T$ **and** $v \notin verts\ T$ **and** $y \in verts\ T$ **and** *pre-digraph.awalk* $T'\ x\ p\ y$
**shows** *awalk* $x\ p\ y$
**proof** −
**have** *0*: $x \in verts\ T\ set\ p \subseteq arcs\ T$
**using** *assms add-leaf-cas-xT-arcsT-if-yT* **unfolding** *pre-digraph.awalk-def* **by** *blast+*
**then show** *?thesis* **using** *add-leaf-awalk-orig-aux assms* **by** *blast*
**qed**

**lemma** *add-leaf-awalk-orig-unique*:
**fixes** $u\ v\ a$
**defines** $T' \equiv (\!|verts = verts\ T \cup \{v\},\ arcs = arcs\ T \cup \{a\},$
$tail = (tail\ T)(a := u),\ head = (head\ T)(a := v)|\!)$
**assumes** $a \notin arcs\ T$ **and** $v \notin verts\ T$ **and** $y \in verts\ T$
**and** *pre-digraph.awalk* $T'\ root\ ps\ y$ **and** *pre-digraph.awalk* $T'\ root\ es\ y$
**shows** $es = ps$
**using** *add-leaf-awalk-orig*[*OF assms*(*2*,*3*)] *assms*(*1*,*4*,*5*,*6*) *unique-awalk* **by** *fast-force*

**lemma** *add-leaf-awalk-new-split'*:
**fixes** $u\ v\ a$
**defines** $T' \equiv (\!|verts = verts\ T \cup \{v\},\ arcs = arcs\ T \cup \{a\},$
$tail = (tail\ T)(a := u),\ head = (head\ T)(a := v)|\!)$
**assumes** $v \notin verts\ T$ **and** $p \neq []$ **and** *pre-digraph.awalk* $T'\ x\ p\ v$
**shows** $\exists as.\ as\ @\ [a] = p$
**using** *assms* **proof**(*induction p arbitrary*: $x$)
**case** (*Cons p ps*)
**then show** *?case*
**proof**(*cases ps* = []))
**case** *True*

```
    then have head T' p = v
    using Cons.prems(3) by (simp add: pre-digraph.awalk-def pre-digraph.cas.simps)
    then have head T p = v ∨ p = a using Cons.hyps(2) by auto
    moreover have p ∈ arcs T ∨ p = a
      using Cons.hyps(2) Cons.prems(3) by (auto simp: pre-digraph.awalk-def)
    ultimately show ?thesis using Cons.prems(1) head-in-verts True by blast
  next
    case False
    then have pre-digraph.cas T' (head T' p) ps v
    using Cons.prems(3) by (simp add: pre-digraph.awalk-def pre-digraph.cas.simps)
    then have pre-digraph.awalk T' (head T' p) ps v
     using Cons.hyps(2) Cons.prems(3) unfolding pre-digraph.awalk-def by auto
    then obtain as where as @ [a] = ps using Cons False by blast
    then show ?thesis by auto
  qed
qed(simp)

lemma add-leaf-awalk-new-split:
  fixes u v a
  defines T' ≡ ⦇verts = verts T ∪ {v}, arcs = arcs T ∪ {a},
              tail = (tail T)(a := u), head = (head T)(a := v)⦈
  assumes v ∉ verts T and u ∈ verts T and p ≠ [] and pre-digraph.awalk T' x
p v
  shows ∃as. as @ [a] = p ∧ pre-digraph.awalk T' x as u
using assms proof(induction p arbitrary: x)
  case (Cons p ps)
  then show ?case
  proof(cases ps = [])
    case True
    then have head T' p = v
    using Cons.prems(4) by (simp add: pre-digraph.awalk-def pre-digraph.cas.simps)
    then have head T p = v ∨ p = a using Cons.hyps(2) by auto
    moreover have p ∈ arcs T ∨ p = a
      using Cons.hyps(2) Cons.prems(4) by (auto simp: pre-digraph.awalk-def)
    ultimately have p = a using Cons.prems(1) by auto
    then have [] @ [a] = p # ps using True by auto
    have tail T' p = u using Cons.hyps(2) ‹p = a› by simp
    then have u = x
    using Cons.prems(4) by (simp add: pre-digraph.awalk-def pre-digraph.cas.simps(2))
    then have pre-digraph.awalk T' x [] u
        using Cons.hyps(2) Cons.prems(2) by (simp add: pre-digraph.awalk-def
pre-digraph.cas.simps)
    then show ?thesis using ‹[] @ [a] = p # ps› by blast
  next
    case False
    then have pre-digraph.cas T' (head T' p) ps v
    using Cons.prems(4) by (simp add: pre-digraph.awalk-def pre-digraph.cas.simps)
    then have pre-digraph.awalk T' (head T' p) ps v
     using Cons.hyps(2) Cons.prems(4) unfolding pre-digraph.awalk-def by auto
```

**then obtain** *as* **where** *as-def*: *as* @ [*a*] = *ps pre-digraph.awalk T′* (*head T′ p*) *as u*
**using** *Cons False* **by** *blast*
**then have** *x* ∈ *verts T′ set* (*p*#*as*) ⊆ *arcs T′ tail T′ p* = *x*
**using** *Cons.prems*(*4*) **by** (*auto simp*: *pre-digraph.awalk-def pre-digraph.cas.simps*)
**then have** *pre-digraph.cas T′ x* (*p*#*as*) *u*
**using** *as-def*(*2*) *pre-digraph.cas.simps*(*2*) **unfolding** *pre-digraph.awalk-def* **by** *fast*
**then have** *pre-digraph.awalk T′ x* (*p*#*as*) *u*
**using** ‹*x* ∈ *verts T′*› ‹*set* (*p*#*as*) ⊆ *arcs T′*› **by** (*simp add*: *pre-digraph.awalk-def*)
**then show** *?thesis* **using** *as-def*(*1*) **by** *auto*
**qed**
**qed**(*simp*)

**lemma** *add-leaf-awalk-new-unique*:
**fixes** *u v a*
**defines** *T′* ≡ (|*verts* = *verts T* ∪ {*v*}, *arcs* = *arcs T* ∪ {*a*},
*tail* = (*tail T*)(*a* := *u*), *head* = (*head T*)(*a* := *v*)|)
**assumes** *a* ∉ *arcs T* **and** *u* ∈ *verts T* **and** *v* ∉ *verts T*
**and** *pre-digraph.awalk T′ root ps v* **and** *pre-digraph.awalk T′ root es v*
**shows** *es* = *ps*
**proof** −
**have** *root* ≠ *v* **using** ‹*v* ∉ *verts T*› *root-in-T* **by** *blast*
**then have** *ps* ≠ [] *es* ≠ []
**using** *assms*(*5*,*6*) *root-in-T pre-digraph.awalk-def pre-digraph.cas.simps*(*1*) **by** *fast*+
**then obtain** *as* **where** *as-def*: *as* @ [*a*] = *ps pre-digraph.awalk T′ root as u*
**using** *add-leaf-awalk-new-split assms*(*1*,*3*−*5*) **by** *blast*
**obtain** *bs* **where** *bs-def*: *bs* @ [*a*] = *es pre-digraph.awalk T′ root bs u*
**using** ‹*es* ≠ []› *add-leaf-awalk-new-split assms*(*1*,*3*,*4*,*6*) **by** *blast*
**then show** *?thesis* **using** *as-def assms*(*1*−*4*) *add-leaf-awalk-orig-unique* **by** *blast*
**qed**

**lemma** *add-leaf-awalk-unique*:
**fixes** *u v a*
**defines** *T′* ≡ (|*verts* = *verts T* ∪ {*v*}, *arcs* = *arcs T* ∪ {*a*},
*tail* = (*tail T*)(*a* := *u*), *head* = (*head T*)(*a* := *v*)|)
**assumes** *a* ∉ *arcs T* **and** *u* ∈ *verts T* **and** *v* ∉ *verts T* **and** *x* ∈ *verts T′*
**shows** ∃!*p*. *pre-digraph.awalk T′ root p x*
**using** *assms add-leaf-awalk-T add-leaf-awalk-T-new*
**by** (*auto simp*: *add-leaf-awalk-new-unique add-leaf-awalk-orig-unique*)

**lemma** *add-leaf-dir-tree*:
⟦*a* ∉ *arcs T*; *u* ∈ *verts T*; *v* ∉ *verts T*⟧
⟹ *directed-tree* (|*verts* = *verts T* ∪ {*v*}, *arcs* = *arcs T* ∪ {*a*},
*tail* = (*tail T*)(*a* := *u*), *head* = (*head T*)(*a* := *v*)|) *root*
**using** *add-leaf-awalk-unique* **by** *unfold-locales* (*auto simp*: *root-in-T*)

**lemma** *add-leaf-dom-preserv*:

$[\![a \notin arcs\ T;\ x \rightarrow_T y]\!]$
$\Longrightarrow x \rightarrow_{(\!|verts = verts\ T \cup \{v\},\ arcs = arcs\ T \cup \{a\},\quad tail = (tail\ T)(a := u),\ head = (head\ T)(a := v)|\!)} y$
**unfolding** *arcs-ends-def arc-to-ends-def* **by** *force*

**end**

## 6.3 Branching Points in Directed Trees

Proofs that show the existence of a last branching point given it is not a chain.

**context** *directed-tree*
**begin**

**lemma** *add-leaf-is-leaf*:
**assumes** $T' = (\!|verts = V,\ arcs = A,\ tail = t,\ head = h|\!)$
**and** $T = (\!|verts = V \cup \{v\},\ arcs = A \cup \{a\},\ tail = t(a := u),\ head = h(a := v)|\!)$
**and** $u \in V$
**and** $v \notin V$
**and** $a \notin A$
**and** *directed-tree* $T'$ *root*$'$
**shows** *leaf* $v$
**proof** −
**have** *0*: *wf-digraph* $T$ **by** (*simp add*: *wf-digraph-axioms*)
**have** *1*: *wf-digraph* $T'$ **using** *assms*(*6*) *directed-tree.axioms*(*1*) **by** *fast*
**then have** $\forall\, a \in arcs\ T.\ tail\ T\ a \neq v$
**by** (*metis Un-insert-right assms*(*1*−*4*) *fun-upd-apply insert-iff*
*pre-digraph.select-convs*(*1*−*3*) *sup-bot-right wf-digraph.tail-in-verts*)
**then have** *out-arcs* $T\ v = \{\}$ **using** *in-out-arcs-conv* **by** *fast*
**moreover have** $v \in verts\ T$ **using** *assms*(*2*) **by** *simp*
**ultimately show** *?thesis* **by** (*simp add*: *leaf-def*)
**qed**

**lemma** *reachable-via-child-impl-same*:
**assumes** $x \rightarrow^*_T v$ **and** $y \rightarrow^*_T v$ **and** $u \rightarrow_T x$ **and** $u \rightarrow_T y$
**shows** $x = y$
**proof** (*rule ccontr*)
**assume** *asm*: $x \neq y$
**obtain** *p1* **where** *p1-def*: *awalk x p1 v* **using** *assms*(*1*) *reachable-awalk* **by** *auto*
**then obtain** *e1* **where** *e1-def*: *awalk u* (*e1*#*p1*) *v* **using** *assms*(*3*) *awalk-Cons-iff* **by** *blast*
**obtain** *p2* **where** *p2-def*: *awalk y p2 v* **using** *assms*(*2*) *reachable-awalk* **by** *auto*
**then obtain** *e2* **where** *e2-def*: *awalk u* (*e2*#*p2*) *v* **using** *assms*(*4*) *awalk-Cons-iff* **by** *blast*
**then have** *e1*#*p1* ≠ *e2*#*p2* **using** *asm awalk-ends p1-def p2-def* **by** *blast*
**then show** *False* **using** *e1-def e2-def unique-awalk-All* **by** *auto*
**qed**

**lemma** *new-leaf-last-in-orig-if-arcs-in-orig*:
**assumes** $x \rightarrow^*_T y$
**and** $T = (\!|verts = V \cup \{v\},\ arcs = A \cup \{a\},\ tail = t(a := u),\ head = h(a := v)|\!)$
**and** $T' = (\!|verts = V,\ arcs = A,\ tail = t,\ head = h|\!)$
**and** $x \in V$
**and** $y \in V$
**and** $u \in V$
**and** $v \notin V$
**and** $a \notin A$
**and** $a1 \in arcs\ T' \wedge a2 \in arcs\ T' \wedge a1 \neq a2 \wedge t\ a1 = y \wedge t\ a2 = y$
**and** *finite* $(arcs\ T)$
**and** $[\![\exists a \in wf\text{-}digraph.branching\text{-}points\ T'.\ x \rightarrow^*_{T'} a;\ directed\text{-}tree\ T'\ r]\!] \Longrightarrow \exists a \in wf\text{-}digraph.last\text{-}branching\text{-}points\ T'.\ x \rightarrow^*_{T'} a$
**and** *directed-tree* $T'\ r$
**shows** $\exists y' \in last\text{-}branching\text{-}points.\ x \rightarrow^*_T y'$
**proof** −
**have** *1*: *wf-digraph* $T'$ **using** *directed-tree.axioms*(*1*) *assms*(*12*) **by** *fast*
**have** $a1 \in arcs\ T' \wedge a2 \in arcs\ T' \wedge a1 \neq a2 \wedge tail\ T'\ a1 = y \wedge tail\ T'\ a2 = y$
**using** *assms*(*3*,*9*) **by** *simp*
**then have** *branching-point*: $y \in wf\text{-}digraph.branching\text{-}points\ T'$
**using** *wf-digraph.branching-points-def 1* **by** *blast*
**then have** $x \rightarrow^*_{T'} y$ **using** *assms*(*1*−*8*,*10*) *1 new-leaf-same-reachables-orig* **by** *blast*
**then have** $\exists a \in wf\text{-}digraph.branching\text{-}points\ T'.\ x \rightarrow^*_{T'} a$ **using** *branching-point* **by** *blast*
**then obtain** *a* **where** *a-def*[*simp*]: $a \in wf\text{-}digraph.last\text{-}branching\text{-}points\ T' \wedge x \rightarrow^*_{T'} a$
**using** *assms*(*11*,*12*) **by** *blast*
**then have** *2*: $a \in wf\text{-}digraph.last\text{-}branching\text{-}points\ T' \wedge x \rightarrow^*_T a$
**using** *new-leaf-same-reachables-new assms*(*2*−*4*,*6*−*8*) *1*
**by** (*metis branch-if-leaf-added new-leaf-no-branch wf-digraph.last-branch-is-branch*)
**have** *3*: $\forall y.\ a \rightarrow^+_T y \longrightarrow a \neq y$ **using** *reachable1-not-reverse* **by** *blast*
**have** $a \in verts\ T'$
**using** *a-def 1* **by** (*simp add*: *wf-digraph.branch-in-verts wf-digraph.last-branch-is-branch*)
**then show** *?thesis*
**using** *new-leaf-last-branch-exists-preserv 1 2 3 assms*(*2*,*3*,*6*−*8*,*10*)
**by** (*metis pre-digraph.select-convs*(*1*,*2*))
**qed**

**lemma** *finite-branch-impl-last-branch*:
**assumes** *finite* $(verts\ T)$
**and** $\exists y \in branching\text{-}points.\ x \rightarrow^*_T y$
**and** *directed-tree* $T\ r$
**shows** $\exists z \in last\text{-}branching\text{-}points.\ x \rightarrow^*_T z$
**using** *assms* **proof**(*induction arbitrary*: *r rule*: *finite-directed-tree-induct*)
**case** (*single-vert t h root*)
**let** *?T* $= (\!|verts = \{root\},\ arcs = \{\},\ tail = t,\ head = h|\!)$
**have** *directed-tree ?T r* **using** *single-vert* **by** *simp*

**then have** *0*: *wf-digraph ?T* **using** *directed-tree.axioms*(*1*) **by** *fast*
**obtain** *y* **where** *y-def*[*simp*]: $y \in$ *wf-digraph.branching-points ?T* $\wedge\ x \rightarrow^*_{?T} y$
**using** *single-vert* **by** *blast*
**have** $y =$ *root*
**by** (*metis y-def empty-iff insert-iff pre-digraph.select-convs*(*1*) *reachable-in-vertsE*)
**then have** $\neg(\exists\, x \in$ *verts ?T*. $x \neq y)$ **by** *simp*
**then have** $\neg(\exists\, x \in$ *wf-digraph.branching-points ?T*. $x \neq y)$
**using** *0 wf-digraph.branch-in-verts* **by** *fast*
**then have** $y \in$ *wf-digraph.last-branching-points ?T*
**using** *wf-digraph.last-branching-points-def 0* **by** *fastforce*
**then show** *?case* **by** *force*
**next**
**case** (*add-leaf T′ V A t h u root a v*)
**let** *?T* = $(\!|$*verts* = $V \cup \{v\}$, *arcs* = $A \cup \{a\}$, *tail* = $t(a := u)$, *head* = $h(a := v)|\!)$
**have** *0*: *wf-digraph ?T* **using** *add-leaf.prems*(*2*) *directed-tree.axioms*(*1*) **by** *fast*
**have** *1*: *wf-digraph T′* **using** *add-leaf.hyps*(*3*) *directed-tree.axioms*(*1*) **by** *fast*
**have** *2*: *finite* (*arcs ?T*)
**using** *directed-tree.verts-finite-imp-arcs-finite add-leaf.hyps*(*1*−*3*) **by** *fastforce*
**obtain** *y* **where** *y-def*[*simp*]: $y \in$ *wf-digraph.branching-points ?T* $\wedge\ x \rightarrow^*_{?T} y$
**using** *add-leaf.prems* **by** *blast*
**then obtain** *a1 a2* **where** *a12*: $a1 \in$*arcs ?T* $\wedge$ $a2 \in$*arcs ?T* $\wedge$ $a1 \neq a2$ $\wedge$ *tail ?T a1* $= y \wedge$ *tail ?T a2* $= y$
**using** *wf-digraph.branching-points-def 0* **by** *blast*
**then have** *y-not-v*: $y \neq v$
**using** *Un-insert-right add-leaf.hyps*(*1*,*3*,*5*) *directed-tree.axioms*(*1*) *fun-upd-apply insert-iff*
**by** (*metis pre-digraph.select-convs*(*1*−*3*) *sup-bot-right wf-digraph.tail-in-verts*)
**have** $y \in$ *verts ?T*
**using** *y-def wf-digraph.branch-in-verts 0* **by** *fast*
**then have** *y-in-T*: $y \in$ *verts T′* **using** *y-not-v add-leaf.hyps*(*1*) **by** *simp*
**have** $x \in$ *verts ?T* **using** *add-leaf.prems*(*1*) *reachable-in-vertsE* **by** *force*
**have** *leaf-v*: *pre-digraph.leaf ?T v*
**using** *directed-tree.add-leaf-is-leaf*[*of ?T*] *add-leaf.hyps*(*1*,*3*−*6*) *add-leaf.prems*(*2*) **by** *blast*
**then have** *out-degree ?T v* $= 0$
**using** *add-leaf.prems*(*2*) *directed-tree.leaf-out-degree-zero* **by** *fast*
**then have** $x \neq v$
**using** *y-not-v y-def 0 Diff-empty add-leaf directed-tree.verts-finite-imp-arcs-finite select-convs*(*1*) *wf-digraph.out-degree-0-only-self* **by** *fastforce*
**then have** *x-in-T′*: $x \in$ *verts T′* **using** ‹$x \in$ *verts ?T*› *add-leaf.hyps*(*1*) **by** *auto*
**show** *?case*
**proof**(*cases a1*=*a* $\vee$ *a2*=*a*)
**case** *True*
**then have** $y = u$ **using** *a12* **by** *fastforce*
**show** *?thesis*
**proof**(*cases* $\exists\, y' \in$*wf-digraph.branching-points ?T*. $y \neq y' \wedge y \rightarrow^*_{?T} y'$)
**case** *True*
**then obtain** *y′* **where** *y′-def*: $y' \in$*wf-digraph.branching-points ?T* $\wedge\ y \neq y'$

$\wedge$ $y \rightarrow^*{}_{?T}$ $y'$
**by** *blast*
**then obtain** *a1 a2* **where** *a12*: *a1∈arcs ?T* $\wedge$ *a2∈arcs ?T* $\wedge$ *a1≠a2* $\wedge$ *tail ?T a1* $= y' \wedge$ *tail ?T a2* $= y'$
**using** *wf-digraph.branching-points-def 0* **by** *blast*
**then have** $y' \neq u$ **using** ‹*y=u*› *y'-def* **by** *blast*
**moreover have** *tail ?T a* $= u$ **by** *simp*
**ultimately have** *a1≠a* $\wedge$ *a2≠a* **using** ‹*y=u*› *a12* **by** *fastforce*
**then have** *3*: *a1∈arcs T'* $\wedge$ *a2∈arcs T'* $\wedge$ *a1≠a2* $\wedge$ *t a1* $= y' \wedge$ *t a2* $= y'$
**using** *a12 add-leaf.hyps*(*1*) **by** *simp*
**then have** *branching-point*: $y' \in$ *wf-digraph.branching-points T'*
**using** *wf-digraph.branching-points-def 1 add-leaf.hyps*(*1*) **by** *fastforce*
**have** *y'-in-T*: $y' \in$ *verts T'* **by** (*simp add*: *1 branching-point wf-digraph.branch-in-verts*)
**have** $x \rightarrow^*{}_{?T}$ $y'$ **using** *y-def y'-def wf-digraph.reachable-trans 0* **by** *fast*
**then show** *?thesis*
**using** *directed-tree.new-leaf-last-in-orig-if-arcs-in-orig*[*of ?T r x y'*]
*add-leaf.prems*(*2*) *2 3 add-leaf.IH add-leaf.hyps*(*1*,*3*−*6*) *x-in-T' y'-in-T*
**by** *simp*
**next**
**case** *False*
**then show** *?thesis* **using** *wf-digraph.last-branching-points-def y-def 0* **by** *fast*
**qed**
**next**
**case** *False*
**then have** *a1∈arcs ?T* $\wedge$ *a2∈arcs ?T* $\wedge$ *a1≠a2* $\wedge$ *t a1* $= y \wedge$ *t a2* $= y$
**using** *a12* **by** *simp*
**then have** *3*: *a1∈arcs T'* $\wedge$ *a2∈arcs T'* $\wedge$ *a1≠a2* $\wedge$ *t a1* $= y \wedge$ *t a2* $= y$
**using** *False a12 add-leaf.hyps*(*1*) **by** *auto*
**have** $x \rightarrow^*{}_{?T}$ $y$ **using** *y-def* **by** *simp*
**then show** *?thesis*
**using** *directed-tree.new-leaf-last-in-orig-if-arcs-in-orig*[*of ?T r x y*]
*add-leaf.prems*(*2*) *2 3 add-leaf.IH add-leaf.hyps*(*1*,*3*−*6*) *x-in-T' y-in-T* **by** *simp*
**qed**
**qed**

**lemma** *subgraph-no-last-branch-chain*:
**assumes** *subgraph C T*
**and** *finite* (*verts T*)
**and** *verts C* $\subseteq$ *verts T* $- \{x.\ \exists\, y \in$ *last-branching-points*. $x \rightarrow^*{}_T y\}$
**shows** *wf-digraph.is-chain C*
**using** *assms finite-branch-impl-last-branch subgraph-no-branch-chain last-branch-is-branch*
**by** (*smt* (*verit*, *ccfv-SIG*) *Collect-cong directed-tree-axioms*)

**lemma** *reach-from-last-in-chain*:
**assumes** $\exists\, y \in$ *last-branching-points*. $y \rightarrow^+{}_T x$
**shows** $x \in$ *verts T* $- \{x.\ \exists\, y \in$ *last-branching-points*. $x \rightarrow^*{}_T y\}$
**using** *assms last-branch-alt reachable1-not-reverse reachable1-reachable reachable1-reachable-trans*

**by** (*smt* (*verit*, *del-insts*) *Diff-iff last-branch-is-branch mem-Collect-eq reachable1-in-verts*(*2*))

Directed Trees don't have merging points.

**lemma** *merging-empty*: *merging-points* = {}
**using** *two-in-arcs-contr merging-points-def* **by** *auto*

**lemma** *subgraph-no-last-merge-chain*:
**assumes** *subgraph C T*
**shows** *wf-digraph.is-chain' C*
**proof** (*rule ccontr*)
**assume** *asm*: ¬*wf-digraph.is-chain' C*
**have** *wf-digraph C* **using** *assms*(*1*) *Digraph-Component.subgraph-def subgraph.sub-G* **by** *auto*
**then obtain** *x* **where** *x-def*: *x* ∈ *wf-digraph.merging-points C*
**using** *wf-digraph.is-chain'-def asm* **by** *blast*
**then have** *x* ∈ *merging-points* **using** *assms*(*1*) *merge-in-supergraph* **by** *simp*
**then show** *False* **using** *merging-empty* **by** *simp*
**qed**

## 6.4 Converting to Trees of Lists

**definition** *to-list-tree* :: ('*a list*, '*b*) *pre-digraph* **where**
*to-list-tree* =
(|*verts* = (λ*x*. [*x*]) ' *verts T*, *arcs* = *arcs T*, *tail* = (λ*x*. [*tail T x*]), *head* = (λ*x*. [*head T x*])|)

**lemma** *to-list-tree-union-verts-eq*: ⋃(*set* ' *verts to-list-tree*) = *verts T*
**using** *to-list-tree-def* **by** *simp*

**lemma** *to-list-tree-cas*: *cas u p v* ⟷ *pre-digraph.cas to-list-tree* [*u*] *p* [*v*]
**by**(*induction p arbitrary*: *u*) (*auto simp*: *Arc-Walk.pre-digraph.cas.simps to-list-tree-def*)

**lemma** *to-list-tree-awalk*: *awalk u p v* ⟷ *pre-digraph.awalk to-list-tree* [*u*] *p* [*v*]
**unfolding** *pre-digraph.awalk-def* **using** *to-list-tree-cas to-list-tree-def* **by** *auto*

**lemma** *to-list-tree-awalk-if-in-verts*:
**assumes** *v* ∈ *verts to-list-tree*
**shows** ∃ *p*. *pre-digraph.awalk to-list-tree* [*root*] *p v*
**proof** −
**have** *root* ∈ *verts T* **using** *root-in-T* **by** *blast*
**obtain** *v'* **where** *0*: *v* = [*v'*] **using** *to-list-tree-def assms*(*1*) **by** *auto*
**then have** *v'* ∈ *verts T* **using** *assms to-list-tree-def* **by** *auto*
**then obtain** *p'* **where** *awalk root p' v'* **using** *unique-awalk* **by** *blast*
**then show** *?thesis* **using** *to-list-tree-awalk 0* **by** *auto*
**qed**

**lemma** *to-list-tree-root-awalk-unique*:
**assumes** *v* ∈ *verts to-list-tree*

**and** *pre-digraph.awalk to-list-tree* [*root*] *p v*
**and** *pre-digraph.awalk to-list-tree* [*root*] *y v*
**shows** $p = y$
**proof** (*rule ccontr*)
**assume** $p \neq y$
**obtain** $v'$ **where** $v'$-*def*: $v = [v']$ **using** *to-list-tree-def assms*(*1*) **by** *auto*
**then have** $v' \in$ *verts T* **using** *assms*(*1*) *to-list-tree-def* **by** *auto*
**show** *False* **using** *to-list-tree-awalk assms* ‹$p \neq y$› *assms*(*2*,*3*) *unique-awalk* $v'$-*def*
**by** *blast*
**qed**

**lemma** *to-list-tree-directed-tree*: *directed-tree to-list-tree* [*root*]
**apply**(*unfold-locales*)
**apply**(*auto simp*: *to-list-tree-def root-in-T*)[*3*]
**by**(*auto intro*: *to-list-tree-awalk-if-in-verts to-list-tree-root-awalk-unique*)

**lemma** *to-list-tree-disjoint-verts*:
⟦$u \in$ *verts to-list-tree*; $v \in$ *verts to-list-tree*; $u \neq v$⟧ $\Longrightarrow$ *set u* $\cap$ *set v* $= \{\}$
**unfolding** *to-list-tree-def* **by** *auto*

**lemma** *to-list-tree-nempty*: $v \in$ *verts to-list-tree* $\Longrightarrow v \neq []$
**unfolding** *to-list-tree-def* **by** *auto*

**lemma** *to-list-tree-single*: $v \in$ *verts to-list-tree* $\Longrightarrow \exists x.\ v = [x] \wedge x \in$ *verts T*
**unfolding** *to-list-tree-def* **by** *auto*

**lemma** *to-list-tree-dom-iff*: $x \rightarrow_T y \longleftrightarrow [x] \rightarrow_{\textit{to-list-tree}} [y]$
**unfolding** *to-list-tree-def arcs-ends-def arc-to-ends-def* **by** *auto*

**end**

**locale** *fin-list-directed-tree* = *finite-directed-tree T* **for** *T* :: (*'a list*,*'b*) *pre-digraph* +
**assumes** *disjoint-verts*: ⟦$u \in$ *verts T*; $v \in$ *verts T*; $u \neq v$⟧ $\Longrightarrow$ *set u* $\cap$ *set v* $= \{\}$
**and** *nempty-verts*: $v \in$ *verts T* $\Longrightarrow v \neq []$

**context** *finite-directed-tree*
**begin**

**lemma** *to-list-tree-fin-digraph*: *fin-digraph to-list-tree*
**by** (*unfold-locales*) (*auto simp*: *to-list-tree-def*)

**lemma** *to-list-tree-finite-directed-tree*: *finite-directed-tree to-list-tree* [*root*]
**by** (*simp add*: *finite-directed-tree-def to-list-tree-fin-digraph to-list-tree-directed-tree*)

**lemma** *to-list-tree-fin-list-directed-tree*: *fin-list-directed-tree* [*root*] *to-list-tree*
**apply**(*simp add*: *fin-list-directed-tree-def to-list-tree-finite-directed-tree*)
**apply**(*unfold-locales*)

**by** (*auto simp*: *to-list-tree-disjoint-verts to-list-tree-nempty*)

**end**

**end**

**theory** *Dtree*
**imports** *Complex-Main Directed-Tree-Additions HOL−Library.FSet*
**begin**

# 7 Algebraic Type for Directed Trees

**datatype** (*dverts*:'*a*, *darcs*: '*b*) *dtree* = *Node* (*root*: '*a*) (*sucs*: (('*a*,'*b*) *dtree* × '*b*) *fset*)

## 7.1 Termination Proofs

**lemma** *fset-sum-ge-elem*: *finite xs* ⟹ *x* ∈ *xs* ⟹ (∑ *u*∈*xs*. (*f*::'*a* ⇒ *nat*) *u*) ≥ *f x*
**by** (*simp add*: *sum-nonneg-leq-bound*)

**lemma** *dtree-size-decr-aux*:
**assumes** (*x*,*y*) ∈ *fset xs*
**shows** *size x* < *size* (*Node r xs*)
**proof** −
**have** *0*: ((*x*,*size x*),*y*) ∈ (*map-prod* (λ*u*. (*u*, *size u*)) (λ*u*. *u*)) ‘ *fset xs* **using** *assms* **by** *fast*
**have** *size x* < *Suc* (*size-prod snd* (λ-. *0*) ((*x*,*size x*),*y*)) **by** *simp*
**also have**
. . . ≤ (∑ *u*∈(*map-prod* (λ*x*. (*x*, *size x*)) (λ*y*. *y*)) ‘ *fset xs*. *Suc* (*size-prod snd* (λ-. *0*) *u*)) + *1*
**using** *fset-sum-ge-elem 0 finite-fset finite-imageI*
**by** (*metis* (*mono-tags*, *lifting*) *add-increasing2 zero-le-one*)
**finally show** *?thesis* **by** *simp*
**qed**

**lemma** *dtree-size-decr-aux'*: *t1* ∈ *fst* ‘ *fset xs* ⟹ *size t1* < *size* (*Node r xs*)
**using** *dtree-size-decr-aux* **by** *fastforce*

**lemma** *dtree-size-decr*[*termination-simp*]:
**assumes** (*x*, *y*) ∈ *fset* (*xs*:: (('*a*, '*b*) *dtree* × '*b*) *fset*)
**shows** *size x* < *Suc* (∑ *u*∈*map-prod* (λ*x*. (*x*, *size x*)) (λ*y*. *y*) ‘ *fset xs*. *Suc* (*Suc* (*snd* (*fst u*))))
**proof** −
**let** *?xs* = (*map-prod* (λ*x*. (*x*, *size x*)) (λ*y*. *y*)) ‘ *fset xs*
**have** *size x* < (∑ *u*∈*?xs*. *Suc* (*size-prod snd* (λ-. *0*) *u*)) + *1*
**using** *dtree-size-decr-aux assms* **by** *fastforce*
**also have** . . . = *Suc* (∑ *u*∈*?xs*. *Suc* (*Suc* (*snd* (*fst u*)))) **by** (*simp add*: *size-prod-simp*)

**finally show** *?thesis* **by** *blast*
**qed**

## 7.2 Dtree Basic Functions

**fun** *darcs-mset* :: ('*a*,'*b*) *dtree* ⇒ '*b multiset* **where**
*darcs-mset* (*Node r xs*) = (∑ (*t*,*e*) ∈ *fset xs*. {#*e*#} + *darcs-mset t*)

**fun** *dverts-mset* :: ('*a*,'*b*) *dtree* ⇒ '*a multiset* **where**
*dverts-mset* (*Node r xs*) = {#*r*#} + (∑ (*t*,*e*) ∈ *fset xs*. *dverts-mset t*)

**abbreviation** *disjoint-darcs* :: (('*a*,'*b*) *dtree* × '*b*) *fset* ⇒ *bool* **where**
*disjoint-darcs xs* ≡ (∀ (*x*,*e1*) ∈ *fset xs*. *e1* ∉ *darcs x* ∧ (∀ (*y*,*e2*) ∈ *fset xs*.
(*darcs x* ∪ {*e1*}) ∩ (*darcs y* ∪ {*e2*}) = {} ∨ (*x*,*e1*)=(*y*,*e2*)))

**fun** *wf-darcs'* :: ('*a*,'*b*) *dtree* ⇒ *bool* **where**
*wf-darcs'* (*Node r xs*) = (*disjoint-darcs xs* ∧ (∀ (*x*,*e*) ∈ *fset xs*. *wf-darcs' x*))

**definition** *wf-darcs* :: ('*a*,'*b*) *dtree* ⇒ *bool* **where**
*wf-darcs t* = (∀ *x* ∈# *darcs-mset t*. *count* (*darcs-mset t*) *x* = *1*)

**fun** *wf-dverts'* :: ('*a*,'*b*) *dtree* ⇒ *bool* **where**
*wf-dverts'* (*Node r xs*) = (∀ (*x*,*e1*) ∈ *fset xs*.
*r* ∉ *dverts x* ∧ (∀ (*y*,*e2*) ∈ *fset xs*. (*dverts x* ∩ *dverts y* = {} ∨ (*x*,*e1*)=(*y*,*e2*)))
∧ *wf-dverts' x*)

**definition** *wf-dverts* :: ('*a*,'*b*) *dtree* ⇒ *bool* **where**
*wf-dverts t* = (∀ *x* ∈# *dverts-mset t*. *count* (*dverts-mset t*) *x* = *1*)

**fun** *dtail* :: ('*a*,'*b*) *dtree* ⇒ ('*b* ⇒ '*a*) ⇒ '*b* ⇒ '*a* **where**
*dtail* (*Node r xs*) *def* = (λ*e*. *if e* ∈ *snd* ‘ *fset xs then r*
*else* (*ffold* (λ(*x*,*e2*) *b*.
*if* (*x*,*e2*) ∉ *fset xs* ∨ *e* ∉ *darcs x* ∨ ¬*wf-darcs* (*Node r xs*)
*then b else dtail x def*) *def xs*) *e*)

**fun** *dhead* :: ('*a*,'*b*) *dtree* ⇒ ('*b* ⇒ '*a*) ⇒ '*b* ⇒ '*a* **where**
*dhead* (*Node r xs*) *def* = (λ*e*. (*ffold* (λ(*x*,*e2*) *b*.
*if* (*x*,*e2*) ∉ *fset xs* ∨ *e* ∉ (*darcs x* ∪ {*e2*}) ∨ ¬*wf-darcs* (*Node r xs*)
*then b else if e*=*e2 then root x else dhead x def e*) (*def e*) *xs*))

**abbreviation** *from-dtree* :: ('*b* ⇒ '*a*) ⇒ ('*b* ⇒ '*a*) ⇒ ('*a*,'*b*) *dtree* ⇒ ('*a*,'*b*)
*pre-digraph* **where**
*from-dtree deft defh t* ≡
(|*verts* = *dverts t*, *arcs* = *darcs t*, *tail* = *dtail t deft*, *head* = *dhead t defh*|)

**abbreviation** *from-dtree'* :: ('*a*,'*b*) *dtree* ⇒ ('*a*,'*b*) *pre-digraph* **where**

*from-dtree′ t* ≡ *from-dtree* (λ-. *root t*) (λ-. *root t*) *t*

**fun** *is-subtree* :: ('*a*,'*b*) *dtree* ⇒ ('*a*,'*b*) *dtree* ⇒ *bool* **where**
*is-subtree x* (*Node r xs*) =
(*x* = *Node r xs* ∨ (∃ (*y*,*e*) ∈ *fset xs*. *is-subtree x y*))

**definition** *strict-subtree* :: ('*a*,'*b*) *dtree* ⇒ ('*a*,'*b*) *dtree* ⇒ *bool* **where**
*strict-subtree t1 t2* ⟷ *is-subtree t1 t2* ∧ *t1* ≠ *t2*

**fun** *num-leaves* :: ('*a*,'*b*) *dtree* ⇒ *nat* **where**
*num-leaves* (*Node r xs*) = (*if xs* = {||} *then 1 else* (∑ (*t*,*e*)∈ *fset xs*. *num-leaves t*))

## 7.3 Dtree Basic Proofs

**lemma** *finite-dverts*: *finite* (*dverts t*)
**by**(*induction t*) *auto*

**lemma** *finite-darcs*: *finite* (*darcs t*)
**by**(*induction t*) *auto*

**lemma** *dverts-child-subseteq*: *x* ∈ *fst* ‘ *fset xs* ⟹ *dverts x* ⊆ *dverts* (*Node r xs*)
**by** *fastforce*

**lemma** *dverts-suc-subseteq*: *x* ∈ *fst* ‘ *fset* (*sucs t*) ⟹ *dverts x* ⊆ *dverts t*
**using** *dverts-child-subseteq*[*of x sucs t root t*] **by** *simp*

**lemma** *dverts-root-or-child*: *v* ∈ *dverts* (*Node r xs*) ⟹ *v* = *r* ∨ *v* ∈ (⋃ (*t*,*e*) ∈ *fset xs*. *dverts t*)
**by** *auto*

**lemma** *dverts-root-or-suc*: *v* ∈ *dverts t* ⟹ *v* = *root t* ∨ (∃ (*t*,*e*) ∈ *fset* (*sucs t*).*v* ∈ *dverts t*)
**using** *dverts-root-or-child*[*of v root t sucs t*] **by** *auto*

**lemma** *dverts-child-if-not-root*:
⟦*v* ∈ *dverts* (*Node r xs*); *v* ≠ *r*⟧ ⟹ ∃ *t*∈*fst* ‘ *fset xs*. *v* ∈ *dverts t*
**by** *force*

**lemma** *dverts-suc-if-not-root*:
⟦*v* ∈ *dverts t*; *v* ≠ *root t*⟧ ⟹ ∃ *t*∈*fst* ‘ *fset* (*sucs t*). *v* ∈ *dverts t*
**using** *dverts-root-or-suc* **by** *force*

**lemma** *darcs-child-subseteq*: *x* ∈ *fst* ‘ *fset xs* ⟹ *darcs x* ⊆ *darcs* (*Node r xs*)
**by** *force*

**lemma** *mset-sum-elem*: *x* ∈# (∑ *y* ∈ *fset Y*. *f y*) ⟹ ∃ *y* ∈ *fset Y*. *x* ∈# *f y*
**by** (*induction Y*) *auto*

**lemma** *mset-sum-elem-iff*: $x \in\# (\sum y \in fset\ Y.\ f\ y) \longleftrightarrow (\exists y \in fset\ Y.\ x \in\# f\ y)$
**by** (*induction Y*) *auto*

**lemma** *mset-sum-elemI*: $[\![y \in fset\ Y;\ x \in\# f\ y]\!] \Longrightarrow x \in\# (\sum y \in fset\ Y.\ f\ y)$
**by** (*induction Y*) *auto*

**lemma** *darcs-mset-elem*:
$x \in\# darcs\text{-}mset\ (Node\ r\ xs) \Longrightarrow \exists (t,e) \in fset\ xs.\ x \in\# darcs\text{-}mset\ t \vee x = e$
**using** *mset-sum-elem* **by** *fastforce*

**lemma** *darcs-mset-if-nsnd*:
$[\![x \in\# darcs\text{-}mset\ (Node\ r\ xs);\ x \notin snd\ `\ fset\ xs]\!] \Longrightarrow \exists (t1,e1) \in fset\ xs.\ x \in\# darcs\text{-}mset\ t1$
**using** *darcs-mset-elem*[*of x r xs*] **by** *force*

**lemma** *darcs-mset-suc-if-nsnd*:
$[\![x \in\# darcs\text{-}mset\ t;\ x \notin snd\ `\ fset\ (sucs\ t)]\!] \Longrightarrow \exists (t1,e1) \in fset\ (sucs\ t).\ x \in\# darcs\text{-}mset\ t1$
**using** *darcs-mset-if-nsnd*[*of x root t sucs t*] **by** *simp*

**lemma** *darcs-mset-if-nchild*:
$[\![x \in\# darcs\text{-}mset\ (Node\ r\ xs);\ \nexists t1\ e1.\ (t1,e1) \in fset\ xs \wedge x \in\# darcs\text{-}mset\ t1]\!]$
$\Longrightarrow x \in snd\ `\ fset\ xs$
**using** *mset-sum-elem* **by** *force*

**lemma** *darcs-mset-if-nsuc*:
$[\![x \in\# darcs\text{-}mset\ t;\ \nexists t1\ e1.\ (t1,e1) \in fset\ (sucs\ t) \wedge x \in\# darcs\text{-}mset\ t1]\!]$
$\Longrightarrow x \in snd\ `\ fset\ (sucs\ t)$
**using** *darcs-mset-if-nchild*[*of x root t sucs t*] **by** *simp*

**lemma** *darcs-mset-if-snd*[*intro*]: $x \in snd\ `\ fset\ xs \Longrightarrow x \in\# darcs\text{-}mset\ (Node\ r\ xs)$
**by** (*induction xs*) *auto*

**lemma** *darcs-mset-suc-if-snd*[*intro*]: $x \in snd\ `\ fset\ (sucs\ t) \Longrightarrow x \in\# darcs\text{-}mset\ t$
**using** *darcs-mset-if-snd*[*of x sucs t root t*] **by** *simp*

**lemma** *darcs-mset-if-child*[*intro*]:
$[\![(t1,e1) \in fset\ xs;\ x \in\# darcs\text{-}mset\ t1]\!] \Longrightarrow x \in\# darcs\text{-}mset\ (Node\ r\ xs)$
**by** (*induction xs*) *auto*

**lemma** *darcs-mset-if-suc*[*intro*]:
$[\![(t1,e1) \in fset\ (sucs\ t);\ x \in\# darcs\text{-}mset\ t1]\!] \Longrightarrow x \in\# darcs\text{-}mset\ t$
**using** *darcs-mset-if-child*[*of t1 e1 sucs t x root t*] **by** *simp*

**lemma** *darcs-mset-sub-darcs*: $set\text{-}mset\ (darcs\text{-}mset\ t) \subseteq darcs\ t$
**proof**(*standard*, *induction t rule*: *darcs-mset.induct*)
**case** (*1 r xs*)

**then show** *?case*
**proof**(*cases x ∈ snd ‘ fset xs*)
**case** *False*
**then obtain** *t1 e1* **where** (*t1*,*e1*) ∈ *fset xs* ∧ *x* ∈# *darcs-mset t1*
**using** *1.prems darcs-mset-if-nsnd*[*of x r*] **by** *blast*
**then show** *?thesis* **using** *1.IH* **by** *force*
**qed**(*force*)
**qed**

**lemma** *darcs-sub-darcs-mset*: *darcs t* ⊆ *set-mset* (*darcs-mset t*)
**proof**(*standard*, *induction t rule*: *darcs-mset.induct*)
**case** (*1 r xs*)
**then show** *?case*
**proof**(*cases x ∈ snd ‘ fset xs*)
**case** *False*
**then obtain** *t1 e1* **where** (*t1*,*e1*) ∈ *fset xs* ∧ *x* ∈ *darcs t1*
**using** *1.prems* **by** *force*
**then show** *?thesis* **using** *1.IH* **by** *blast*
**qed**(*blast*)
**qed**

**lemma** *darcs-mset-eq-darcs*[*simp*]: *set-mset* (*darcs-mset t*) = *darcs t*
**using** *darcs-mset-sub-darcs darcs-sub-darcs-mset* **by** *force*

**lemma** *dverts-mset-elem*:
*x* ∈# *dverts-mset* (*Node r xs*) ⟹ (∃ (*t*,*e*) ∈ *fset xs*. *x* ∈# *dverts-mset t*) ∨ *x* = *r*
**using** *mset-sum-elem* **by** *fastforce*

**lemma** *dverts-mset-if-nroot*:
⟦*x* ∈# *dverts-mset* (*Node r xs*); *x* ≠ *r*⟧ ⟹ ∃ (*t1*,*e1*) ∈ *fset xs*. *x* ∈# *dverts-mset t1*
**using** *dverts-mset-elem*[*of x r xs*] **by** *blast*

**lemma** *dverts-mset-suc-if-nroot*:
⟦*x* ∈# *dverts-mset t*; *x* ≠ *root t*⟧ ⟹ ∃ (*t1*,*e1*) ∈ *fset* (*sucs t*). *x* ∈# *dverts-mset t1*
**using** *dverts-mset-if-nroot*[*of x root t sucs t*] **by** *simp*

**lemma** *dverts-mset-if-nchild*:
⟦*x* ∈# *dverts-mset* (*Node r xs*); ∄ *t1 e1*. (*t1*,*e1*) ∈ *fset xs* ∧ *x* ∈# *dverts-mset t1*⟧ ⟹ *x* = *r*
**using** *mset-sum-elem* **by** *force*

**lemma** *dverts-mset-if-nsuc*:
⟦*x* ∈# *dverts-mset t*; ∄ *t1 e1*. (*t1*,*e1*) ∈ *fset* (*sucs t*) ∧ *x* ∈# *dverts-mset t1*⟧ ⟹ *x* = *root t*
**using** *dverts-mset-if-nchild*[*of x root t sucs t*] **by** *simp*

**lemma** *dverts-mset-if-root*[*intro*]: $x = r \Longrightarrow x \in\# \ dverts\text{-}mset\ (Node\ r\ xs)$
**by** *simp*

**lemma** *dverts-mset-suc-if-root*[*intro*]: $x = root\ t \Longrightarrow x \in\# \ dverts\text{-}mset\ t$
**using** *dverts-mset-if-root*[*of x root t sucs t*] **by** *simp*

**lemma** *dverts-mset-if-child*[*intro*]:
$[\![(t1,e1) \in fset\ xs;\ x \in\# \ dverts\text{-}mset\ t1]\!] \Longrightarrow x \in\# \ dverts\text{-}mset\ (Node\ r\ xs)$
**by** (*induction xs*) *auto*

**lemma** *dverts-mset-if-suc*[*intro*]:
$[\![(t1,e1) \in fset\ (sucs\ t);\ x \in\# \ dverts\text{-}mset\ t1]\!] \Longrightarrow x \in\# \ dverts\text{-}mset\ t$
**using** *dverts-mset-if-child*[*of t1 e1 sucs t x root t*] **by** *simp*

**lemma** *dverts-mset-sub-dverts*: $set\text{-}mset\ (dverts\text{-}mset\ t) \subseteq dverts\ t$
**proof**(*standard*, *induction t*)
**case** (*Node r xs*)
**then show** *?case*
**proof**(*cases* $x = r$)
**case** *False*
**then obtain** *t1 e1* **where** $(t1,e1) \in fset\ xs \wedge x \in\# \ dverts\text{-}mset\ t1$
**using** *Node.prems dverts-mset-if-nroot* **by** *fast*
**then show** *?thesis* **using** *Node.IH* **by** *fastforce*
**qed**(*simp*)
**qed**

**lemma** *dverts-sub-dverts-mset*: $dverts\ t \subseteq set\text{-}mset\ (dverts\text{-}mset\ t)$
**proof**(*standard*, *induction t rule*: *dverts-mset.induct*)
**case** (*1 r xs*)
**then show** *?case*
**proof**(*cases* $x = r$)
**case** *False*
**then obtain** *t1 e1* **where** $(t1,e1) \in fset\ xs \wedge x \in dverts\ t1$
**using** *1.prems* **by** *force*
**then show** *?thesis* **using** *1.IH* **by** *blast*
**qed**(*simp*)
**qed**

**lemma** *dverts-mset-eq-dverts*[*simp*]: $set\text{-}mset\ (dverts\text{-}mset\ t) = dverts\ t$
**using** *dverts-mset-sub-dverts dverts-sub-dverts-mset* **by** *force*

**lemma** *mset-sum-count-le*: $y \in fset\ Y \Longrightarrow count\ (f\ y)\ x \leq count\ (\sum y \in fset\ Y.\ f\ y)\ x$
**by** (*induction Y*) *auto*

**lemma** *darcs-mset-alt*:
$darcs\text{-}mset\ (Node\ r\ xs) = (\sum (t,e) \in fset\ xs.\ \{\#e\#\}) + (\sum (t,e) \in fset\ xs.\ darcs\text{-}mset\ t)$
**by** (*induction xs*) *auto*

**lemma** *darcs-mset-ge-child*:
*t1* ∈ *fst* ‘ *fset xs* ⟹ *count* (*darcs-mset t1*) *x* ≤ *count* (*darcs-mset* (*Node r xs*)) *x*
**by** (*induction xs*) *force+*

**lemma** *darcs-mset-ge-suc*:
*t1* ∈ *fst* ‘ *fset* (*sucs t*) ⟹ *count* (*darcs-mset t1*) *x* ≤ *count* (*darcs-mset t*) *x*
**using** *darcs-mset-ge-child*[*of t1 sucs t x root t*] **by** *simp*

**lemma** *darcs-mset-count-sum-aux*:
(∑ (*t1*,*e1*) ∈ *fset xs*. *count* (*darcs-mset t1*) *x*) = *count* ((∑ (*t*,*e*) ∈ *fset xs*. *darcs-mset t*)) *x*
**by** (*smt* (*verit*, *ccfv-SIG*) *count-add-mset count-sum multi-self-add-other-not-self prod.case prod.case-distrib split-cong sum.cong*)

**lemma** *darcs-mset-count-sum-aux0*:
*x* ∉ *snd* ‘ *fset xs* ⟹ *count* ((∑ (*t*, *e*)∈*fset xs*. {#*e*#})) *x* = *0*
**by** (*induction xs*) *auto*

**lemma** *darcs-mset-count-sum-eq*:
*x* ∉ *snd* ‘ *fset xs*
⟹ (∑ (*t1*,*e1*) ∈ *fset xs*. *count* (*darcs-mset t1*) *x*) = *count* (*darcs-mset* (*Node r xs*)) *x*
**unfolding** *darcs-mset-alt* **using** *darcs-mset-count-sum-aux darcs-mset-count-sum-aux0*
**by** *fastforce*

**lemma** *darcs-mset-count-sum-ge*:
(∑ (*t1*,*e1*) ∈ *fset xs*. *count* (*darcs-mset t1*) *x*) ≤ *count* (*darcs-mset* (*Node r xs*)) *x*
**by** (*induction xs*) (*auto split*: *prod.splits*)

**lemma** *wf-darcs-alt*: *wf-darcs t* ⟷ (∀ *x*. *count* (*darcs-mset t*) *x* ≤ *1*)
**unfolding** *wf-darcs-def* **by** (*metis count-greater-eq-one-iff dual-order.eq-iff linorder-le-cases*)

**lemma** *disjoint-darcs-simp*:
⟦(*t1*,*e1*) ∈ *fset xs*; (*t2*,*e2*) ∈ *fset xs*; (*t1*,*e1*) ≠ (*t2*,*e2*); *disjoint-darcs xs*⟧
⟹ (*darcs t1* ∪ {*e1*}) ∩ (*darcs t2* ∪ {*e2*}) = {}
**by** *fast*

**lemma** *disjoint-darcs-single*: *e* ∉ *darcs t* ⟷ *disjoint-darcs* {|(*t*,*e*)|}
**by** *simp*

**lemma** *disjoint-darcs-insert*: *disjoint-darcs* (*finsert x xs*) ⟹ *disjoint-darcs xs*
**by** *simp fast*

**lemma** *wf-darcs-rec*[*dest*]:
**assumes** *wf-darcs* (*Node r xs*) **and** *t1* ∈ *fst* ‘ *fset xs*
**shows** *wf-darcs t1*
**unfolding** *wf-darcs-def* **proof** (*rule ccontr*)

**assume** *asm*: ¬ (∀ *x* ∈# *darcs-mset t1*. *count* (*darcs-mset t1*) *x* = *1*)
**then obtain** *x* **where** *x-def*: *x* ∈# *darcs-mset t1 count* (*darcs-mset t1*) *x* ≠ *1*
**by** *blast*
**then have** *count* (*darcs-mset t1*) *x* > *1* **by** (*simp add*: *order-le-neq-trans*)
**then have** *count* (*darcs-mset* (*Node r xs*)) *x* > *1*
**using** *assms*(*2*) *darcs-mset-ge-child*[*of t1 xs x*] **by** *simp*
**moreover have** *x* ∈# (*darcs-mset* (*Node r xs*))
**using** *x-def*(*1*) *assms*(*2*) **by** *fastforce*
**ultimately show** *False* **using** *assms*(*1*) **unfolding** *wf-darcs-def* **by** *simp*
**qed**

**lemma** *disjoint-darcs-if-wf-aux1*: ⟦*wf-darcs* (*Node r xs*); (*t1*,*e1*) ∈ *fset xs*⟧ ⟹ *e1* ∉ *darcs t1*
**apply** (*induction xs*)
**apply**(*auto simp*: *wf-darcs-def split*: *if-splits prod.splits*)[*2*]
**by** (*metis UnI2 add-is-1 count-eq-zero-iff*)

**lemma** *fset-sum-ge-elem2*:
⟦*x* ∈ *fset X*; *y* ∈ *fset X*; *x* ≠ *y*⟧ ⟹ (*f* :: *'a* ⇒ *nat*) *x* + *f y* ≤ (∑ *x* ∈ *fset X*. *f x*)
**by** (*induction X*) (*auto simp*: *fset-sum-ge-elem*)

**lemma** *darcs-children-count-ge2-aux*:
**assumes** (*t1*,*e1*) ∈ *fset xs* **and** (*t2*,*e2*) ∈ *fset xs* **and** (*t1*,*e1*) ≠ (*t2*,*e2*)
**and** *e* ∈ *darcs t1* **and** *e* ∈ *darcs t2*
**shows** (∑ (*t1*, *e1*)∈*fset xs*. *count* (*darcs-mset t1*) *e*) ≥ *2*
**proof** −
**have** *2* ≤ *1* + *count* (*darcs-mset t2*) *e*
**using** *assms*(*2*,*5*) **by** *simp*
**also have** ... ≤ *count* (*darcs-mset t1*) *e* + *count* (*darcs-mset t2*) *e*
**using** *assms*(*1*,*4*) **by** *simp*
**finally show** *?thesis*
**using** *fset-sum-ge-elem2*[*OF assms*(*1*−*3*), *of* λ(*t1*,*e1*). *count* (*darcs-mset t1*) *e*] **by** *simp*
**qed**

**lemma** *darcs-children-count-ge2*:
**assumes** (*t1*,*e1*) ∈ *fset xs* **and** (*t2*,*e2*) ∈ *fset xs* **and** (*t1*,*e1*) ≠ (*t2*,*e2*)
**and** *e* ∈ *darcs t1* **and** *e* ∈ *darcs t2*
**shows** *count* (*darcs-mset* (*Node r xs*)) *e* ≥ *2*
**using** *darcs-children-count-ge2-aux*[*OF assms*] *darcs-mset-count-sum-ge dual-order.trans*
**by** *fast*

**lemma** *darcs-children-count-not1*:
⟦(*t1*,*e1*) ∈ *fset xs*; (*t2*,*e2*) ∈ *fset xs*; (*t1*,*e1*) ≠ (*t2*,*e2*); *e* ∈ *darcs t1*; *e* ∈ *darcs t2*⟧
⟹ *count* (*darcs-mset* (*Node r xs*)) *e* ≠ *1*
**using** *darcs-children-count-ge2* **by** *fastforce*

**lemma** *disjoint-darcs-if-wf-aux2*:
**assumes** *wf-darcs* (*Node r xs*)
**and** (*t1*,*e1*) ∈ *fset xs* **and** (*t2*,*e2*) ∈ *fset xs* **and** (*t1*,*e1*) ≠ (*t2*,*e2*)
**shows** *darcs t1* ∩ *darcs t2* = {}
**proof**(*rule ccontr*)
**assume** *darcs t1* ∩ *darcs t2* ≠ {}
**then obtain** *e* **where** *e-def*: *e* ∈ *darcs t1 e* ∈ *darcs t2* **by** *blast*
**then have** *e* ∈ *darcs* (*Node r xs*) **using** *assms*(*2*) **by** *force*
**then have** *e* ∈# *darcs-mset* (*Node r xs*) **using** *darcs-mset-eq-darcs* **by** *fast*
**then show** *False*
**using** *darcs-children-count-ge2*[*OF assms*(*2*−*4*) *e-def*] *assms*(*1*) **unfolding** *wf-darcs-def* **by** *simp*
**qed**

**lemma** *darcs-child-count-ge1*:
⟦(*t1*,*e1*) ∈ *fset xs*; *e2* ∈ *darcs t1*⟧ ⟹ *count* (∑ (*t*, *e*)∈*fset xs*. *darcs-mset t*) *e2* ≥ *1*
**by** (*simp add*: *mset-sum-elemI*)

**lemma** *darcs-snd-count-ge1*:
(*t2*,*e2*) ∈ *fset xs* ⟹ *count* (∑ (*t*, *e*)∈*fset xs*. {#*e*#}) *e2* ≥ *1*
**by** (*simp add*: *mset-sum-elemI*)

**lemma** *darcs-child-count-ge2*:
⟦(*t1*,*e1*) ∈ *fset xs*; (*t2*,*e2*) ∈ *fset xs*; *e2* ∈ *darcs t1*⟧ ⟹ *count* (*darcs-mset* (*Node r xs*)) *e2* ≥ *2*
**unfolding** *darcs-mset-alt*
**by** (*metis darcs-child-count-ge1 darcs-snd-count-ge1 add-mono count-union one-add-one*)

**lemma** *disjoint-darcs-if-wf-aux3*:
**assumes** *wf-darcs* (*Node r xs*) **and** (*t1*,*e1*) ∈ *fset xs* **and** (*t2*,*e2*) ∈ *fset xs*
**shows** *e2* ∉ *darcs t1*
**proof**
**assume** *asm*: *e2* ∈ *darcs t1*
**then have** *e2* ∈ *darcs* (*Node r xs*) **using** *assms*(*2*) **by** *force*
**then have** *e2* ∈# *darcs-mset* (*Node r xs*) **using** *darcs-mset-eq-darcs* **by** *fast*
**then show** *False* **using** *darcs-child-count-ge2 asm assms*(*1*−*3*) **unfolding** *wf-darcs-def* **by** *fastforce*
**qed**

**lemma** *darcs-snds-count-ge2-aux*:
**assumes** (*t1*,*e1*) ∈ *fset xs* **and** (*t2*,*e2*) ∈ *fset xs* **and** (*t1*,*e1*) ≠ (*t2*,*e2*) **and** *e1* = *e2*
**shows** *count* (∑ (*t*, *e*)∈*fset xs*. {#*e*#}) *e2* ≥ *2*
**using** *assms* **proof**(*induction xs*)
**case** (*insert x xs*)
**then consider** *x* = (*t1*,*e1*) | *x* = (*t2*,*e2*) | (*t1*,*e1*) ∈ *fset xs* (*t2*,*e2*) ∈ *fset xs* **by** *auto*
**then show** *?case*

```
  proof(cases)
    case 1
    then have count (∑(t, e)∈fset xs. {#e#}) e2 ≥ 1
      using insert.prems(2,3) darcs-snd-count-ge1 by auto
    then show ?thesis using insert.prems(4) insert.hyps 1 by auto
  next
    case 2
    then have count (∑(t, e)∈fset xs. {#e#}) e2 ≥ 1
      using insert.prems(1,3,4) darcs-snd-count-ge1 by auto
    then show ?thesis using insert.prems(4) insert.hyps 2 by auto
  next
    case 3
    then show ?thesis using insert.IH insert.prems(3,4) insert.hyps by auto
  qed
qed(simp)

lemma darcs-snds-count-ge2:
  ⟦(t1,e1) ∈ fset xs; (t2,e2) ∈ fset xs; (t1,e1) ≠ (t2,e2); e1 = e2⟧
    ⟹ count (darcs-mset (Node r xs)) e2 ≥ 2
  using darcs-snds-count-ge2-aux unfolding darcs-mset-alt by fastforce

lemma disjoint-darcs-if-wf-aux4:
  assumes wf-darcs (Node r xs)
      and (t1,e1) ∈ fset xs
      and (t2,e2) ∈ fset xs
      and (t1,e1) ≠ (t2,e2)
    shows e1 ≠ e2
proof
  assume asm: e1 = e2
  have e2 ∈# darcs-mset (Node r xs) using assms(3) darcs-mset-if-snd by fast-
force
  then show False
   using assms(1) darcs-snds-count-ge2[OF assms(2−4) asm] unfolding wf-darcs-def
by simp
qed

lemma disjoint-darcs-if-wf-aux5:
  ⟦wf-darcs (Node r xs); (t1,e1) ∈ fset xs; (t2,e2) ∈ fset xs; (t1,e1) ≠ (t2,e2)⟧
    ⟹(darcs t1 ∪ {e1}) ∩ (darcs t2 ∪ {e2}) = {}
 by (auto dest: disjoint-darcs-if-wf-aux4 disjoint-darcs-if-wf-aux3 disjoint-darcs-if-wf-aux2)

lemma disjoint-darcs-if-wf-xs: wf-darcs (Node r xs) ⟹ disjoint-darcs xs
  by (auto dest: disjoint-darcs-if-wf-aux1 disjoint-darcs-if-wf-aux5)

lemma disjoint-darcs-if-wf: wf-darcs t ⟹ disjoint-darcs (sucs t)
  using disjoint-darcs-if-wf-xs[of root t sucs t] by simp

lemma wf-darcs'-if-darcs: wf-darcs t ⟹ wf-darcs' t
proof(induction t)
```

**case** (*Node r xs*)
**then show** *?case* **using** *disjoint-darcs-if-wf-xs*[*OF Node.prems*] **by** *fastforce*
**qed**

**lemma** *wf-darcs-if-darcs'-aux*:
⟦∀ (*x*,*e*) ∈ *fset xs*. *wf-darcs x*; *disjoint-darcs xs*⟧ ⟹ *wf-darcs* (*Node r xs*)
**apply**(*simp split*: *prod.splits*)
**apply**(*induction xs*)
**apply**(*auto simp*: *wf-darcs-def count-eq-zero-iff*)[*2*]
**by** (*fastforce dest*: *mset-sum-elem*)+

**lemma** *wf-darcs-if-darcs'*: *wf-darcs' t* ⟹ *wf-darcs t*
**proof**(*induction t*)
**case** (*Node r xs*)
**then show** *?case* **using** *wf-darcs-if-darcs'-aux*[*of xs*] **by** *fastforce*
**qed**

**corollary** *wf-darcs-iff-darcs'*: *wf-darcs t* ⟷ *wf-darcs' t*
**using** *wf-darcs-if-darcs' wf-darcs'-if-darcs* **by** *blast*

**lemma** *disjoint-darcs-subset*:
**assumes** *xs* |⊆| *ys* **and** *disjoint-darcs ys*
**shows** *disjoint-darcs xs*
**proof** (*rule ccontr*)
**assume** ¬ *disjoint-darcs xs*
**then obtain** *x e1 y e2* **where** *x-def*: (*x*,*e1*) ∈ *fset xs* (*y*,*e2*) ∈ *fset xs*
*e1* ∈ *darcs x* ∨ (*darcs x* ∪ {*e1*}) ∩ (*darcs y* ∪ {*e2*}) ≠ {} ∧ (*x*,*e1*)≠(*y*,*e2*)
**by** *blast*
**have** (*x*,*e1*) ∈ *fset ys* (*y*,*e2*) ∈ *fset ys* **using** *x-def*(*1*,*2*) *assms*(*1*) *less-eq-fset.rep-eq*
**by** *fast*+
**then show** *False* **using** *assms*(*2*) *x-def*(*3*) **by** *fast*
**qed**

**lemma** *disjoint-darcs-img*:
**assumes** *disjoint-darcs xs* **and** ∀ (*t*,*e*) ∈ *fset xs*. *darcs* (*f t*) ⊆ *darcs t*
**shows** *disjoint-darcs* ((λ(*t*,*e*). (*f t*,*e*)) |‘| *xs*) (**is** *disjoint-darcs ?xs*)
**proof** (*rule ccontr*)
**assume** ¬ *disjoint-darcs ?xs*
**then obtain** *x1 e1 y1 e2* **where** *asm*: (*x1*,*e1*) ∈ *fset ?xs* (*y1*,*e2*) ∈ *fset ?xs*
*e1* ∈ *darcs x1* ∨ (*darcs x1* ∪ {*e1*}) ∩ (*darcs y1* ∪ {*e2*}) ≠ {} ∧ (*x1*,*e1*)≠(*y1*,*e2*)
**by** *blast*
**then obtain** *x2* **where** *x2-def*: *f x2* = *x1* (*x2*,*e1*) ∈ *fset xs* **by** *auto*
**obtain** *y2* **where** *y2-def*: *f y2* = *y1* (*y2*,*e2*) ∈ *fset xs* **using** *asm*(*2*) **by** *auto*
**have** *darcs x1* ⊆ *darcs x2* **using** *assms*(*2*) *x2-def* **by** *fast*
**moreover have** *darcs y1* ⊆ *darcs y2* **using** *assms*(*2*) *y2-def* **by** *fast*
**ultimately have** ¬ *disjoint-darcs xs* **using** *asm*(*3*) *x2-def y2-def* **by** *fast*
**then show** *False* **using** *assms*(*1*) **by** *blast*
**qed**

**lemma** *dverts-mset-count-sum-ge*:
(∑ (*t1*,*e1*) ∈ *fset xs*. *count* (*dverts-mset t1*) *x*) ≤ *count* (*dverts-mset* (*Node r xs*)) *x*
**by** (*induction xs*) *auto*

**lemma** *dverts-children-count-ge2-aux*:
**assumes** (*t1*,*e1*) ∈ *fset xs* **and** (*t2*,*e2*) ∈ *fset xs* **and** (*t1*,*e1*) ≠ (*t2*,*e2*)
**and** *x* ∈ *dverts t1* **and** *x* ∈ *dverts t2*
**shows** (∑ (*t1*, *e1*)∈*fset xs*. *count* (*dverts-mset t1*) *x*) ≥ *2*
**proof** −
**have** *2* ≤ *count* (*dverts-mset t1*) *x* + *1* **using** *assms*(*4*) **by** *simp*
**also have** ... ≤ *count* (*dverts-mset t1*) *x* + *count* (*dverts-mset t2*) *x* **using** *assms*(*5*) **by** *simp*
**finally show** *?thesis*
**using** *fset-sum-ge-elem2*[*OF assms*(*1*−*3*), *of* λ(*t1*,*e1*). *count* (*dverts-mset t1*) *x*] **by** *simp*
**qed**

**lemma** *dverts-children-count-ge2*:
**assumes** (*t1*,*e1*) ∈ *fset xs* **and** (*t2*,*e2*) ∈ *fset xs* **and** (*t1*,*e1*) ≠ (*t2*,*e2*)
**and** *x* ∈ *dverts t1* **and** *x* ∈ *dverts t2*
**shows** *count* (*dverts-mset* (*Node r xs*)) *x* ≥ *2*
**using** *dverts-children-count-ge2-aux*[*OF assms*] *dverts-mset-count-sum-ge le-trans* **by** *fast*

**lemma** *disjoint-dverts-if-wf-aux*:
**assumes** *wf-dverts* (*Node r xs*)
**and** (*t1*,*e1*) ∈ *fset xs* **and** (*t2*,*e2*) ∈ *fset xs* **and** (*t1*,*e1*) ≠ (*t2*,*e2*)
**shows** *dverts t1* ∩ *dverts t2* = {}
**proof** (*rule ccontr*)
**assume** *dverts t1* ∩ *dverts t2* ≠ {}
**then obtain** *x* **where** *x-def*: *x* ∈ *dverts t1 x* ∈ *dverts t2* **by** *blast*
**then have** *2* ≤ *count* (*dverts-mset* (*Node r xs*)) *x*
**using** *dverts-children-count-ge2*[*OF assms*(*2*−*4*)] **by** *blast*
**moreover have** *x* ∈# (*dverts-mset* (*Node r xs*)) **using** *assms*(*2*) *x-def*(*1*) **by** *fastforce*
**ultimately show** *False* **using** *assms*(*1*) **unfolding** *wf-dverts-def* **by** *fastforce*
**qed**

**lemma** *disjoint-dverts-if-wf*:
*wf-dverts* (*Node r xs*)
⟹ ∀ (*x*,*e1*) ∈ *fset xs*. ∀ (*y*,*e2*) ∈ *fset xs*. (*dverts x* ∩ *dverts y* = {} ∨ (*x*,*e1*)=(*y*,*e2*))
**using** *disjoint-dverts-if-wf-aux* **by** *fast*

**lemma** *disjoint-dverts-if-wf-sucs*:
*wf-dverts t*
⟹ ∀ (*x*,*e1*) ∈ *fset* (*sucs t*). ∀ (*y*,*e2*) ∈ *fset* (*sucs t*).
(*dverts x* ∩ *dverts y* = {} ∨ (*x*,*e1*)=(*y*,*e2*))

**using** *disjoint-dverts-if-wf*[*of root t sucs t*] **by** *simp*

**lemma** *dverts-child-count-ge1*:
⟦(*t1*,*e1*) ∈ *fset xs*; *x* ∈ *dverts t1*⟧ ⟹ *count* (∑(*t*, *e*)∈*fset xs*. *dverts-mset t*) *x* ≥ *1*
**by** (*simp add*: *mset-sum-elemI*)

**lemma** *root-not-child-if-wf-dverts*: ⟦*wf-dverts* (*Node r xs*); (*t1*,*e1*) ∈ *fset xs*⟧ ⟹ *r* ∉ *dverts t1*
**by** (*fastforce dest*: *dverts-child-count-ge1 simp*: *wf-dverts-def*)

**lemma** *root-not-child-if-wf-dverts'*: *wf-dverts* (*Node r xs*) ⟹ ∀(*t1*,*e1*) ∈ *fset xs*. *r* ∉ *dverts t1*
**by** (*fastforce dest*: *dverts-child-count-ge1 simp*: *wf-dverts-def*)

**lemma** *dverts-mset-ge-child*:
*t1* ∈ *fst* ‘ *fset xs* ⟹ *count* (*dverts-mset t1*) *x* ≤ *count* (*dverts-mset* (*Node r xs*)) *x*
**by** (*induction xs*) *force*+

**lemma** *wf-dverts-rec*[*dest*]:
**assumes** *wf-dverts* (*Node r xs*) **and** *t1* ∈ *fst* ‘ *fset xs*
**shows** *wf-dverts t1*
**unfolding** *wf-dverts-def* **proof** (*rule ccontr*)
**assume** *asm*: ¬ (∀ *x* ∈# *dverts-mset t1*. *count* (*dverts-mset t1*) *x* = *1*)
**then obtain** *x* **where** *x-def*: *x* ∈# *dverts-mset t1 count* (*dverts-mset t1*) *x* ≠ *1*
**by** *blast*
**then have** *count* (*dverts-mset t1*) *x* > *1* **by** (*simp add*: *order-le-neq-trans*)
**then have** *count* (*dverts-mset* (*Node r xs*)) *x* > *1*
**using** *assms*(*2*) *dverts-mset-ge-child*[*of t1 xs x r*] **by** *simp*
**moreover have** *x* ∈# (*dverts-mset* (*Node r xs*))
**using** *x-def*(*1*) *assms*(*2*) **by** *fastforce*
**ultimately show** *False* **using** *assms*(*1*) **unfolding** *wf-dverts-def* **by** *fastforce*
**qed**

**lemma** *wf-dverts'-if-dverts*: *wf-dverts t* ⟹ *wf-dverts' t*
**proof**(*induction t*)
**case** (*Node r xs*)
**then have** ∀(*x*,*e1*)∈*fset xs*. *wf-dverts' x* **by** *auto*
**then show** *?case*
**using** *disjoint-dverts-if-wf*[*OF Node.prems*] *root-not-child-if-wf-dverts'*[*OF Node.prems*]
**by** *fastforce*
**qed**

**lemma** *wf-dverts-if-dverts'-aux*:
⟦∀(*x*,*e*) ∈ *fset xs*. *wf-dverts x*;
∀(*x*,*e1*) ∈ *fset xs*. *r* ∉ *dverts x* ∧ (∀(*y*,*e2*) ∈ *fset xs*.
(*dverts x* ∩ *dverts y* = {} ∨ (*x*,*e1*)=(*y*,*e2*)))⟧
⟹ *wf-dverts* (*Node r xs*)

```
  apply(simp split: prod.splits)
  apply(induction xs)
   apply(auto simp: wf-dverts-def count-eq-zero-iff)[2]
  by (fastforce dest: mset-sum-elem)+

lemma wf-dverts-if-dverts': wf-dverts' t ⟹ wf-dverts t
proof(induction t)
  case (Node r xs)
  then show ?case using wf-dverts-if-dverts'-aux[of xs] by fastforce
qed

corollary wf-dverts-iff-dverts': wf-dverts t ⟷ wf-dverts' t
  using wf-dverts-if-dverts' wf-dverts'-if-dverts by blast

lemma wf-dverts-sub:
  assumes xs |⊆| ys and wf-dverts (Node r ys)
  shows wf-dverts (Node r xs)
proof −
  have ys |∪| xs = ys using assms(1) by blast
  then have wf-dverts (Node r (ys |∪| xs)) using assms(2) by simp
  then show ?thesis unfolding wf-dverts-iff-dverts' by fastforce
qed

lemma count-subset-le:
  xs |⊆| ys ⟹ count (∑ x ∈ fset xs. f x) a ≤ count (∑ x ∈ fset ys. f x) a
proof(induction ys arbitrary: xs)
  case (insert y ys)
  then show ?case
  proof(cases y |∈| xs)
    case True
    then obtain xs' where xs'-def: finsert y xs' = xs y |∉| xs'
      by blast
    then have xs' |⊆| ys using insert.prems by blast
    have count (∑ x ∈ fset xs. f x) a = count (∑ x ∈ fset xs'. f x) a + count (f y)
a
      using xs'-def by auto
    also have ... ≤ count (∑ x ∈ fset ys. f x) a + count (f y) a
      using ‹xs' |⊆| ys› insert.IH by simp
    also have ... = count (∑ x ∈ fset (finsert y ys). f x) a
      using insert.hyps by auto
    finally show ?thesis .
  next
    case False
    then have count (∑ x ∈ fset xs. f x) a ≤ count (∑ x ∈ fset ys. f x) a
      using insert.prems insert.IH by blast
    then show ?thesis using insert.hyps by auto
  qed
qed(simp)
```

**lemma** *darcs-mset-count-le-subset*:
*xs* $|\subseteq|$ *ys* $\Longrightarrow$ *count* (*darcs-mset* (*Node r' xs*)) *x* $\leq$ *count* (*darcs-mset* (*Node r ys*)) *x*
**using** *count-subset-le* **by** *fastforce*

**lemma** *wf-darcs-sub*: $[\![$*xs* $|\subseteq|$ *ys*; *wf-darcs* (*Node r' ys*)$]\!] \Longrightarrow$ *wf-darcs* (*Node r xs*)
**unfolding** *wf-darcs-def* **using** *darcs-mset-count-le-subset*
**by** (*smt* (*verit*, *best*) *count-greater-eq-one-iff le-trans verit-la-disequality*)

**lemma** *wf-darcs-sucs*: $[\![$*wf-darcs t*; *x* $\in$ *fset* (*sucs t*)$]\!] \Longrightarrow$ *wf-darcs* (*Node r* {|*x*|})
**using** *wf-darcs-sub*[*of* {|*x*|} *sucs t root t*] **by** (*simp add*: *less-eq-fset.rep-eq*)

**lemma** *size-fset-alt*:
*size-fset* (*size-prod snd* (λ-. *0*)) (*map-prod* (λ*t*. (*t*, *size t*)) (λ*x*. *x*) |‘| *xs*)
= ($\sum$ (*x*,*y*)$\in$ *fset xs*. *size x* + *2*)
**proof** −
**have** *size-fset* (*size-prod snd* (λ-. *0*)) (*map-prod* (λ*t*. (*t*, *size t*)) (λ*x*. *x*) |‘| *xs*)
= ($\sum$ *u*$\in$(λ(*x*,*y*). ((*x*,*size x*), *y*)) ‘ *fset xs*. *snd* (*fst u*) + *2*)
**by** (*simp add*: *size-prod-simp map-prod-def*)
**also have** ... = ($\sum$ (*x*,*y*) $\in$ *fset xs*. *size x* + *2*)
**using** *case-prod-beta'* *comm-monoid-add-class.sum.eq-general*
**by** (*smt* (*verit*, *del-insts*) *Pair-inject fstI imageE imageI prod-eqI snd-conv*)
**finally show** *?thesis* **.**
**qed**

**lemma** *dtree-size-alt*: *size* (*Node r xs*) = ($\sum$ (*x*,*y*)$\in$ *fset xs*. *size x* + *2*) + *1*
**using** *size-fset-alt* **by** *auto*

**lemma** *dtree-size-eq-root*: *size* (*Node r xs*) = *size* (*Node r' xs*)
**by** *auto*

**lemma** *size-combine-decr*: *size* (*Node* (*r*@*root t1*) (*sucs t1*)) < *size* (*Node r* {|(*t1*, *e1*)|})
**using** *dtree-size-eq-root*[*of r*@*root t1 sucs t1 root t1*] **by** *simp*

**lemma** *size-le-if-child-subset*: *xs* $|\subseteq|$ *ys* $\Longrightarrow$ *size* (*Node r xs*) $\leq$ *size* (*Node v ys*)
**unfolding** *dtree-size-alt* **by** (*simp add*: *dtree-size-alt less-eq-fset.rep-eq sum.subset-diff*)

**lemma** *size-le-if-sucs-subset*: *sucs t1* $|\subseteq|$ *sucs t2* $\Longrightarrow$ *size t1* $\leq$ *size t2*
**using** *size-le-if-child-subset*[*of sucs t1 sucs t2 root t1 root t2*] **by** *simp*

**lemma** *combine-uneq*: *Node r* {|(*t1*, *e1*)|} $\neq$ *Node* (*r*@*root t1*) (*sucs t1*)
**using** *size-combine-decr*[*of r t1 e1*] **by** *fastforce*

**lemma** *child-uneq*: *t* $\in$ *fst* ‘ *fset xs* $\Longrightarrow$ *Node r xs* $\neq$ *t*
**using** *dtree-size-decr-aux'* **by** *fastforce*

**lemma** *suc-uneq*: *t1* $\in$ *fst* ‘ *fset* (*sucs t*) $\Longrightarrow$ *t* $\neq$ *t1*
**using** *child-uneq*[*of t1 sucs t root t*] **by** *simp*

**lemma** *singleton-uneq*: *Node r* {|(*t*,*e*)|} ≠ *t*
**using** *child-uneq*[*of t*] **by** *simp*

**lemma** *child-uneq′*: *t* ∈ *fst* ‘ *fset xs* ⟹ *Node r xs* ≠ *Node v* (*sucs t*)
**using** *dtree-size-decr-aux′*[*of t*] *dtree-size-eq-root*[*of root t sucs t*] **by** *auto*

**lemma** *suc-uneq′*: *t1* ∈ *fst* ‘ *fset* (*sucs t*) ⟹ *t* ≠ *Node v* (*sucs t1*)
**using** *child-uneq′*[*of t1 sucs t root t*] **by** *simp*

**lemma** *singleton-uneq′*: *Node r* {|(*t*,*e*)|} ≠ *Node v* (*sucs t*)
**using** *child-uneq′*[*of t*] **by** *simp*

**lemma** *singleton-suc*: *t* ∈ *fst* ‘ *fset* (*sucs* (*Node r* {|(*t*,*e*)|}))
**by** *simp*

**lemma** *fcard-image-le*: *fcard* (*f* |‘| *xs*) ≤ *fcard xs*
**by** (*simp add*: *FSet.fcard.rep-eq card-image-le*)

**lemma** *sum-img-le*:
**assumes** ∀ *t* ∈ *fst* ‘ *fset xs*. (*g*::′*a* ⇒ *nat*) (*f t*) ≤ *g t*
**shows** (∑ (*x*,*y*)∈ *fset* ((λ(*t*,*e*). (*f t*, *e*)) |‘| *xs*). *g x*) ≤ (∑ (*x*,*y*)∈ *fset xs*. *g x*)
**using** *assms* **proof**(*induction xs*)
**case** (*insert x xs*)
**obtain** *t e* **where** *t-def*: *x* = (*t*,*e*) **by** *fastforce*
**then show** *?case*
**proof**(*cases* (*f t*,*e*) ∉ *fset* ((λ(*t*,*e*). (*f t*, *e*)) |‘| *xs*))
**case** *True*
**then have** (∑ (*x*,*y*)∈ *fset* ((λ(*t*,*e*). (*f t*, *e*)) |‘| (*finsert x xs*)). *g x*)
= *g* (*f t*) + (∑ (*x*,*y*)∈ *fset* ((λ(*t*,*e*). (*f t*, *e*)) |‘| *xs*). *g x*)
**using** *t-def* **by** *auto*
**also have** ... ≤ *g t* + (∑ (*x*,*y*)∈ *fset* ((λ(*t*,*e*). (*f t*, *e*)) |‘| *xs*). *g x*)
**using** *insert.prems t-def* **by** *auto*
**also have** ... ≤ *g t* + (∑ (*x*,*y*)∈ *fset xs*. *g x*) **using** *insert* **by** *simp*
**finally show** *?thesis* **using** *insert.hyps t-def* **by** *fastforce*
**next**
**case** *False*
**then have** (∑ (*x*,*y*)∈ *fset* ((λ(*t*,*e*). (*f t*, *e*)) |‘| (*finsert x xs*)). *g x*)
= (∑ (*x*,*y*)∈ *fset* ((λ(*t*,*e*). (*f t*, *e*)) |‘| *xs*). *g x*)
**by** (*metis* (*no-types*, *lifting*) *t-def fimage-finsert finsert-absorb prod.case*)
**also have** ... ≤ (∑ (*x*,*y*)∈ *fset xs*. *g x*) **using** *insert* **by** *simp*
**finally show** *?thesis* **using** *insert.hyps t-def* **by** *fastforce*
**qed**
**qed** (*simp*)

**lemma** *dtree-size-img-le*:
**assumes** ∀ *t* ∈ *fst* ‘ *fset xs*. *size* (*f t*) ≤ *size t*
**shows** *size* (*Node r* ((λ(*t*,*e*). (*f t*, *e*)) |‘| *xs*)) ≤ *size* (*Node r xs*)
**using** *sum-img-le*[*of xs* λ*x*. *size x* + *2*] *dtree-size-alt assms*

**by** (*metis* (*mono-tags*, *lifting*) *add-right-mono*)

**lemma** *sum-img-lt*:
**assumes** $\forall\, t \in fst\ `\ fset\ xs.\ (g::'a \Rightarrow nat)\ (f\ t) \leq g\ t$
**and** $\exists\, t \in fst\ `\ fset\ xs.\ g\ (f\ t) < g\ t$
**and** $\forall\, t \in fst\ `\ fset\ xs.\ g\ t > 0$
**shows** $(\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs).\ g\ x) < (\sum (x,y) \in fset\ xs.\ g\ x)$
**using** *assms* **proof**(*induction xs*)
**case** (*insert x xs*)
**obtain** *t e* **where** *t-def*: $x = (t,e)$ **by** *fastforce*
**then show** *?case*
**proof**(*cases* $(f\ t,e) \notin fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs)$)
**case** *f-notin-xs*: *True*
**show** *?thesis*
**proof**(*cases g* $(f\ t) < g\ t$)
**case** *True*
**have** $(\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ (finsert\ x\ xs)).\ g\ x)$
$= g\ (f\ t) + (\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs).\ g\ x)$
**using** *t-def f-notin-xs* **by** *auto*
**also have** $\ldots < g\ t + (\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs).\ g\ x)$
**using** *True* **by** *simp*
**also have** $\ldots \leq g\ t + (\sum (x,y) \in fset\ xs.\ g\ x)$ **using** *sum-img-le insert.prems*(*1*)
**by** *auto*
**finally show** *?thesis* **using** *insert.hyps t-def* **by** *fastforce*
**next**
**case** *False*
**then have** *0*: $\exists\, t \in fst\ `\ fset\ xs.\ g\ (f\ t) < g\ t$ **using** *insert.prems*(*2*) *t-def* **by**
*simp*
**have** $(\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ (finsert\ x\ xs)).\ g\ x)$
$= g\ (f\ t) + (\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs).\ g\ x)$
**using** *t-def f-notin-xs* **by** *auto*
**also have** $\ldots \leq g\ t + (\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs).\ g\ x)$
**using** *t-def insert.prems*(*1*) **by** *simp*
**also have** $\ldots < g\ t + (\sum (x,y) \in fset\ xs.\ g\ x)$ **using** *insert.IH insert.prems*(*1*,*3*)
*0* **by** *simp*
**finally show** *?thesis* **using** *insert.hyps t-def* **by** *fastforce*
**qed**
**next**
**case** *False*
**then have** $(\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ (finsert\ x\ xs)).\ g\ x)$
$= (\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs).\ g\ x)$
**by** (*metis* (*no-types*, *lifting*) *t-def fimage-finsert finsert-absorb prod.case*)
**also have** $\ldots \leq (\sum (x,y) \in fset\ xs.\ g\ x)$ **using** *sum-img-le insert.prems*(*1*) **by**
*auto*
**also have** $\ldots < g\ t + (\sum (x,y) \in fset\ xs.\ g\ x)$ **using** *insert.prems*(*3*) *t-def* **by**
*simp*
**finally show** *?thesis* **using** *insert.hyps t-def* **by** *fastforce*
**qed**
**qed** (*simp*)

**lemma** *dtree-size-img-lt*:
**assumes** $\forall\, t \in fst\ `\ fset\ xs.\ size\ (f\ t) \leq size\ t$
**and** $\exists\, t \in fst\ `\ fset\ xs.\ size\ (f\ t) < size\ t$
**shows** $size\ (Node\ r\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs)) < size\ (Node\ r\ xs)$
**proof** −
**have** *0*: $\forall\, t \in fst\ `\ fset\ xs.\ size\ (f\ t) + 2 \leq size\ t + 2$ **using** *assms(1)* **by** *simp*
**have** $\forall\, t \in fst\ `\ fset\ xs.\ 0 < size\ t + 2$ **by** *simp*
**then show** *?thesis* **using** *sum-img-lt*[*OF 0*] *dtree-size-alt assms(2)* **by** (*smt* (*z3*) *add-less-mono1*)
**qed**

**lemma** *sum-img-eq*:
**assumes** $\forall\, t \in fst\ `\ fset\ xs.\ (g::'a \Rightarrow nat)\ (f\ t) = g\ t$
**and** $fcard\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs) = fcard\ xs$
**shows** $(\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs).\ g\ x) = (\sum (x,y) \in fset\ xs.\ g\ x)$
**using** *assms* **proof**(*induction xs*)
**case** (*insert x xs*)
**obtain** *t e* **where** *t-def*: $x = (t,e)$ **by** *fastforce*
**then have** *0*: $(f\ t,e) \notin fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs)$
**using** *insert.prems(2) insert.hyps fcard-finsert-if fcard-image-le*
**by** (*metis* (*mono-tags*, *lifting*) *case-prod-conv fimage-finsert leD lessI*)
**then have** *1*: $fcard\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs) = fcard\ xs$
**using** *insert.prems(2) insert.hyps t-def Suc-inject*
**by** (*metis* (*mono-tags*, *lifting*) *fcard-finsert-if fimage-finsert old.prod.case*)
**have** $(\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ (finsert\ x\ xs)).\ g\ x)$
$= g\ (f\ t) + (\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs).\ g\ x)$
**using** *t-def 0* **by** *auto*
**also have** $\ldots = g\ t + (\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs).\ g\ x)$
**using** *insert.prems t-def* **by** *auto*
**also have** $\ldots = g\ t + (\sum (x,y) \in fset\ xs.\ g\ x)$ **using** *insert.IH 1 insert.prems(1)*
**by** *simp*
**finally show** *?case* **using** *insert.hyps t-def* **by** *fastforce*
**qed** (*simp*)

**lemma** *elem-neq-if-fset-neq*:
$((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs) \neq xs \Longrightarrow \exists\, t \in fst\ `\ fset\ xs.\ f\ t \neq t$
**by** (*smt* (*verit*, *ccfv-threshold*) *case-prod-eta case-prod-eta fimage.rep-eq fset-inject fst-conv*
*image-cong image-ident image-subset-iff old.prod.case prod.case-distrib split-cong subsetI*)

**lemma** *ffold-commute-supset*:
$[\![xs\ |\subseteq|\ ys;\ P\ ys;\ \bigwedge ys\ xs.\ [\![xs\ |\subseteq|\ ys;\ P\ ys]\!] \Longrightarrow P\ xs;$
$\bigwedge xs.\ comp\text{-}fun\text{-}commute\ (\lambda a\ b.\ if\ a \notin fset\ xs \vee \neg Q\ a\ b \vee \neg P\ xs\ then\ b\ else\ R\ a\ b)]\!]$
$\Longrightarrow ffold\ (\lambda a\ b.\ if\ a \notin fset\ ys \vee \neg Q\ a\ b \vee \neg P\ ys\ then\ b\ else\ R\ a\ b)\ acc\ xs$
$= ffold\ (\lambda a\ b.\ if\ a \notin fset\ xs \vee \neg Q\ a\ b \vee \neg P\ xs\ then\ b\ else\ R\ a\ b)\ acc\ xs$
**proof**(*induction xs arbitrary*: *ys*)

**case** *empty*
**show** *?case*
**unfolding** *empty.prems*(*4*)[*THEN comp-fun-commute.ffold-empty*]
**by** *simp*
**next**
**case** (*insert x xs*)
**let** *?f* = λ*a b*. *if a* ∉ *fset ys* ∨ ¬*Q a b* ∨ ¬*P ys then b else R a b*
**let** *?f'* = λ*a b*. *if a* ∉ *fset xs* ∨ ¬*Q a b* ∨ ¬*P xs then b else R a b*
**let** *?f1* = λ*a b*. *if a* ∉ *fset* (*finsert x xs*) ∨ ¬*Q a b* ∨ ¬*P* (*finsert x xs*) *then b else R a b*
**have** *0*: *P* (*finsert x xs*) **using** *insert.prems* **by** *simp*
**have** *1*: *xs* |⊆| (*finsert x xs*) **by** *blast*
**have** *2*: *comp-fun-commute ?f1* **using** *insert.prems*(*4*) **by** *blast*
**have** *3*: *x* ∈ *fset ys* **using** *insert.prems*(*1*) **by** *fastforce*
**have** *ffold ?f acc* (*finsert x xs*) = *?f x* (*ffold ?f acc xs*)
**using** *comp-fun-commute.ffold-finsert*[*of ?f*] *insert.prems*(*4*) *insert.hyps* **by** *blast*
**also have** ... = *?f x* (*ffold ?f' acc xs*) **using** *insert.IH*[*of ys*] *insert.prems* **by** *fastforce*
**also have** ... = *?f x* (*ffold ?f1 acc xs*) **using** *insert.IH*[*OF 1 0*] *insert.prems*(*3*,*4*) **by** *presburger*
**also have** ... = *?f1 x* (*ffold ?f1 acc xs*) **using** *0 3 insert.prems*(*2*) **by** *fastforce*
**also have** ... = *ffold ?f1 acc* (*finsert x xs*)
**using** *comp-fun-commute.ffold-finsert*[*of ?f1 x xs*] *2 insert.hyps* **by** *presburger*
**finally show** *?case* **.**
**qed**

**lemma** *ffold-eq-fold*: ⟦*finite xs*; *f* = *g*⟧ ⟹ *ffold f acc* (*Abs-fset xs*) = *Finite-Set.fold g acc xs*
**unfolding** *ffold-def* **by** (*simp add*: *Abs-fset-inverse*)

**lemma** *Abs-fset-sub-if-sub*:
**assumes** *finite ys* **and** *xs* ⊆ *ys*
**shows** *Abs-fset xs* |⊆| *Abs-fset ys*
**proof** (*rule ccontr*)
**assume** ¬(*Abs-fset xs* |⊆| *Abs-fset ys*)
**then obtain** *x* **where** *x-def*: *x* |∈| *Abs-fset xs x* |∉| *Abs-fset ys* **by** *blast*
**then have** *x* ∈ *fset* (*Abs-fset xs*) ∧ *x* ∉ *fset* (*Abs-fset ys*) **by** *fast*
**moreover have** *finite xs* **using** *assms finite-subset* **by** *auto*
**ultimately show** *False* **using** *assms Abs-fset-inverse* **by** *blast*
**qed**

**lemma** *fold-commute-supset*:
**assumes** *finite ys* **and** *xs* ⊆ *ys* **and** *P ys* **and** ⋀*ys xs*. ⟦*xs* ⊆ *ys*; *P ys*⟧ ⟹ *P xs*
**and** ⋀*xs*. *comp-fun-commute* (λ*a b*. *if a* ∉ *xs* ∨ ¬*Q a b* ∨ ¬*P xs then b else R a b*)
**shows** *Finite-Set.fold* (λ*a b*. *if a* ∉ *ys* ∨ ¬*Q a b* ∨ ¬*P ys then b else R a b*) *acc xs*
= *Finite-Set.fold* (λ*a b*. *if a* ∉ *xs* ∨ ¬*Q a b* ∨ ¬*P xs then b else R a b*) *acc*

*xs*
**proof** −
**let** *?f* = λ*a b*. *if a* ∉ *ys* ∨ ¬*Q a b* ∨ ¬*P ys then b else R a b*
**let** *?f′* = λ*a b*. *if a* ∉ *xs* ∨ ¬*Q a b* ∨ ¬*P xs then b else R a b*
**let** *?P* = λ*xs*. *P* (*fset xs*)
**let** *?g* = λ*a b*. *if a* ∉ *fset* (*Abs-fset ys*) ∨ ¬*Q a b* ∨ ¬(*?P* (*Abs-fset ys*)) *then b else R a b*
**let** *?g′* = λ*a b*. *if a* ∉ *fset* (*Abs-fset xs*) ∨ ¬*Q a b* ∨ ¬(*?P* (*Abs-fset xs*)) *then b else R a b*
**have** *0*: *finite xs* **using** *assms*(*1*,*2*) *finite-subset* **by** *auto*
**then have** *1*: *Abs-fset xs* |⊆| (*Abs-fset ys*) **using** *Abs-fset-sub-if-sub*[*OF assms*(*1*,*2*)] **by** *blast*
**have** *2*: *?P* (*Abs-fset ys*) **by** (*simp add*: *Abs-fset-inverse assms*(*1*,*3*))
**have** *3*: ⋀*ys xs*. ⟦*xs* |⊆| *ys*; *?P ys*⟧ ⟹ *?P xs* **by** (*simp add*: *assms*(*4*) *less-eq-fset.rep-eq*)
**have** *4*: ⋀*xs*. *comp-fun-commute* (λ*a b*. *if a* ∉ *fset xs* ∨ ¬*Q a b* ∨ ¬(*?P xs*) *then b else R a b*)
**using** *assms*(*5*) **by** (*simp add*: *less-eq-fset.rep-eq*)
**have** *?f′* = *?g′* **by** (*simp add*: *Abs-fset-inverse 0*)
**have** *?f* = *?g* **by** (*simp add*: *Abs-fset-inverse assms*(*1*))
**then have** *Finite-Set.fold* (λ*a b*. *if a* ∉ *ys* ∨ ¬*Q a b* ∨ ¬*P ys then b else R a b*) *acc xs*
= *ffold ?g acc* (*Abs-fset xs*) **by** (*simp add*: *0 ffold-eq-fold*)
**also have** … = *ffold ?g′ acc* (*Abs-fset xs*)
**using** *ffold-commute-supset*[*OF 1*, *of ?P*, *OF 2 3 4*] **by** *simp*
**finally show** *?thesis* **using** ‹*?f′* = *?g′*› **by** (*simp add*: *0 ffold-eq-fold*)
**qed**

**lemma** *dtail-commute-aux*:
**fixes** *r xs e def*
**defines** *f* ≡ (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset xs* ∨ *e* ∉ *darcs x* ∨ ¬*wf-darcs* (*Node r xs*)
*then b else dtail x def*)
**shows** (*f y* ∘ *f x*) *z* = (*f x* ∘ *f y*) *z*
**proof** −
**obtain** *y1 y2* **where** *y-def*: *y* = (*y1*,*y2*) **by** *fastforce*
**obtain** *x1 x2* **where** *x-def*: *x* = (*x1*,*x2*) **by** *fastforce*
**show** *?thesis*
**proof**(*cases* (*x1*,*x2*) ∈ *fset xs* ∧ (*y1*,*y2*) ∈ *fset xs*)
**case** *0*: *True*
**then show** *?thesis*
**proof**(*cases e* ∈ *darcs x1* ∧ *e* ∈ *darcs y1*)
**case** *True*
**then have** *1*: *x1* = *y1* ∨ ¬*wf-darcs* (*Node r xs*) **using** *0 disjoint-darcs-if-wf-aux2* **by** *fast*
**then show** *?thesis* **using** *assms* **by** (*cases x1*=*y1*)(*auto simp*: *x-def y-def*)
**next**
**case** *False*
**then show** *?thesis* **using** *assms* **by** (*simp add*: *x-def y-def*)
**qed**

**next**
**case** *False*
**then show** *?thesis* **using** *assms* **by** (*simp add*: *x-def y-def*)
**qed**
**qed**

**lemma** *dtail-commute*:
*comp-fun-commute* (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset xs* ∨ *e* ∉ *darcs x* ∨ ¬*wf-darcs* (*Node r xs*)
*then b else dtail x def*)
**using** *dtail-commute-aux*[*of xs*] **by** *unfold-locales blast*

**lemma** *dtail-f-alt*:
**assumes** *P* = (λ*xs*. *wf-darcs* (*Node r xs*))
**and** *Q* = (λ(*t1*,*e1*) *b*. *e* ∈ *darcs t1*)
**and** *R* = (λ(*t1*,*e1*) *b*. *dtail t1 def*)
**shows** (λ(*t1*,*e1*) *b*. *if* (*t1*,*e1*) ∉ *fset xs* ∨ *e* ∉ *darcs t1*∨ ¬*wf-darcs* (*Node r xs*)
*then b else dtail t1 def*)
= (λ*a b*. *if a* ∉ *fset xs* ∨ ¬ *Q a b* ∨ ¬ *P xs then b else R a b*)
**using** *assms* **by** *fast*

**lemma** *dtail-f-alt-commute*:
**assumes** *P* = (λ*xs*. *wf-darcs* (*Node r xs*))
**and** *Q* = (λ(*t1*,*e1*) *b*. *e* ∈ *darcs t1*)
**and** *R* = (λ(*t1*,*e1*) *b*. *dtail t1 def*)
**shows** *comp-fun-commute* (λ*a b*. *if a* ∉ *fset xs* ∨ ¬ *Q a b* ∨ ¬ *P xs then b else R a b*)
**using** *dtail-commute*[*of xs e r def*] *dtail-f-alt*[*OF assms*] **by** *simp*

**lemma** *dtail-ffold-supset*:
**assumes** *xs* |⊆| *ys* **and** *wf-darcs* (*Node r ys*)
**shows** *ffold* (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset ys* ∨ *e* ∉ *darcs x* ∨ ¬*wf-darcs* (*Node r ys*)
*then b else dtail x def*) *def xs*
= *ffold* (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset xs* ∨ *e* ∉ *darcs x* ∨ ¬*wf-darcs* (*Node r xs*)
*then b else dtail x def*) *def xs*
**proof** −
**let** *?P* = λ*xs*. *wf-darcs* (*Node r xs*)
**let** *?Q* = λ(*t1*,*e1*) *b*. *e* ∈ *darcs t1*
**let** *?R* = λ(*t1*,*e1*) *b*. *dtail t1 def*
**have** *0*: ⋀*xs*. *comp-fun-commute* (λ*a b*. *if a* ∉ *fset xs* ∨ ¬ *?Q a b* ∨ ¬ *?P xs then b else ?R a b*)
**using** *dtail-f-alt-commute* **by** *fast*
**have** *ffold* (λ*a b*. *if a* ∉ *fset ys* ∨ ¬ *?Q a b* ∨ ¬ *?P ys then b else ?R a b*) *def xs*
= *ffold* (λ*a b*. *if a* ∉ *fset xs* ∨ ¬ *?Q a b* ∨ ¬ *?P xs then b else ?R a b*) *def xs*
**using** *ffold-commute-supset*[*OF assms*(*1*),*of ?P ?Q ?R*,*OF assms*(*2*) *wf-darcs-sub 0*] **by** *simp*
**then show** *?thesis* **using** *dtail-f-alt*[*of ?P r ?Q e ?R*] **by** *simp*
**qed**

**lemma** *dtail-in-child-eq-child-ffold*:
**assumes** (*t*,*e1*) ∈ *fset xs* **and** *e* ∈ *darcs t* **and** *wf-darcs* (*Node r xs*)
**shows** *ffold* (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset xs* ∨ *e* ∉ *darcs x* ∨ ¬*wf-darcs* (*Node r xs*)
*then b else dtail x def*) *def xs*
= *dtail t def*
**using** *assms* **proof**(*induction xs*)
**case** (*insert x′ xs*)
**let** *?f* = (λ(*x*,*e2*) *b*.
*if* (*x*,*e2*) ∉ *fset* (*finsert x′ xs*) ∨ *e* ∉ *darcs x* ∨ ¬*wf-darcs* (*Node r* (*finsert x′ xs*))
*then b else dtail x def*)
**let** *?f′* = (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset xs* ∨ *e* ∉ *darcs x* ∨ ¬*wf-darcs* (*Node r xs*)
*then b else dtail x def*)
**obtain** *x e3* **where** *x-def*: *x′* = (*x*,*e3*) **by** *fastforce*
**show** *?case*
**proof**(*cases x*=*t*)
**case** *True*
**have** *ffold ?f def* (*finsert x′ xs*) = (*?f x′* (*ffold ?f def xs*))
**using** *comp-fun-commute.ffold-finsert*[*of ?f x′ xs def*] *dtail-commute insert.hyps*
**by** *fast*
**also have** ... = (*?f* (*x*,*e3*) (*ffold ?f def xs*)) **using** *x-def* **by** *blast*
**also have** ... = *dtail x def* **using** *x-def insert.prems*(*2*,*3*) *True* **by** *fastforce*
**finally show** *?thesis* **using** *True* **by** *blast*
**next**
**case** *False*
**then have** *0*: (*t*,*e1*) ∈ *fset xs* **using** *insert.prems*(*1*) *x-def* **by** *simp*
**have** *1*: *wf-darcs* (*Node r xs*) **using** *wf-darcs-sub*[*OF fsubset-finsertI insert.prems*(*3*)]
.
**have** *2*: *xs* |⊆| (*finsert x′ xs*) **by** *blast*
**have** (*x*,*e3*) ∈ *fset* (*finsert x′ xs*) **using** *x-def* **by** *simp*
**have** *3*: *e* ∉ *darcs x* **using** *insert.prems*(*1*−*3*) *disjoint-darcs-if-wf x-def False*
**by** *fastforce*
**have** *ffold ?f def* (*finsert x′ xs*) = (*?f x′* (*ffold ?f def xs*))
**using** *comp-fun-commute.ffold-finsert*[*of ?f x′ xs def*] *dtail-commute insert.hyps*
**by** *fast*
**also have** ... = (*?f* (*x*,*e3*) (*ffold ?f def xs*)) **using** *x-def* **by** *blast*
**also have** ... = (*ffold ?f def xs*) **using** *3* **by** *fastforce*
**also have** ... = (*ffold ?f′ def xs*)
**using** *dtail-ffold-supset*[*of xs finsert x′ xs*] *insert.prems*(*3*) *2* **by** *simp*
**also have** ... = *dtail t def* **using** *insert.IH 0 1 insert.prems*(*2*) **by** *fast*
**finally show** *?thesis* **.**
**qed**
**qed**(*simp*)

**lemma** *dtail-in-child-eq-child*:
**assumes** (*t*,*e1*) ∈ *fset xs* **and** *e* ∈ *darcs t* **and** *wf-darcs* (*Node r xs*)
**shows** *dtail* (*Node r xs*) *def e* = *dtail t def e*

**using** *assms dtail-in-child-eq-child-ffold*[*OF assms*] *disjoint-darcs-if-wf-aux3* **by** *fastforce*

**lemma** *dtail-ffold-notelem-eq-def*:
**assumes** $\forall (t,e1) \in fset\ xs.\ e \notin darcs\ t$
**shows** *ffold* ($\lambda(x,e2)$ *b. if* $(x,e2) \notin fset\ ys \vee e \notin darcs\ x \vee \neg wf\text{-}darcs\ (Node\ r\ ys)$
*then b else dtail x def*) *def xs* = *def*
**using** *assms* **proof**(*induction xs*)
**case** *empty*
**show** *?case*
**unfolding** *dtail-commute*[*THEN comp-fun-commute.ffold-empty*]
**by** *simp*
**next**
**case** (*insert x' xs*)
**obtain** *x e3* **where** *x-def*: $x' = (x,e3)$ **by** *fastforce*
**let** *?f* = ($\lambda(x,e2)$ *b. if* $(x,e2) \notin fset\ ys \vee e \notin darcs\ x \vee \neg wf\text{-}darcs\ (Node\ r\ ys)$
*then b else dtail x def*)
**have** *ffold ?f def* (*finsert x' xs*) = *?f x'* (*ffold ?f def xs*)
**using** *comp-fun-commute.ffold-finsert*[*of ?f x' xs*] *dtail-commute insert.hyps* **by** *fast*
**also have** ... = (*ffold ?f def xs*) **using** *insert.prems* **by** *auto*
**also have** ... = *def* **using** *insert.IH insert.prems* **by** *simp*
**finally show** *?case* **.**
**qed**

**lemma** *dtail-notelem-eq-def*:
**assumes** $e \notin darcs\ t$
**shows** *dtail t def e* = *def e*
**proof** −
**obtain** *r xs* **where** *xs-def*[*simp*]: *t* = *Node r xs* **using** *dtree.exhaust* **by** *auto*
**let** *?f* = ($\lambda(x,e2)$ *b. if* $(x,e2) \notin fset\ xs \vee e \notin darcs\ x \vee \neg wf\text{-}darcs\ (Node\ r\ xs)$
*then b else dtail x def*)
**have** *0*: $\forall (t,\ e1) \in fset\ xs.\ e \notin darcs\ t$ **using** *assms* **by** *auto*
**have** *dtail* (*Node r xs*) *def e* = *ffold ?f def xs e* **using** *assms* **by** *auto*
**then show** *?thesis* **using** *dtail-ffold-notelem-eq-def 0* **by** *fastforce*
**qed**

**lemma** *dhead-commute-aux*:
**fixes** *r xs e def*
**defines** $f \equiv (\lambda(x,e2)$ *b. if* $(x,e2) \notin fset\ xs \vee e \notin (darcs\ x \cup \{e2\}) \vee \neg wf\text{-}darcs\ (Node\ r\ xs)$
*then b else if e=e2 then root x else dhead x def e*)
**shows** (*f y* ∘ *f x*) *z* = (*f x* ∘ *f y*) *z*
**proof** −
**obtain** *x1 x2* **where** *x-def*: *x* = (*x1*,*x2*) **by** *fastforce*
**obtain** *y1 y2* **where** *y-def*: *y* = (*y1*,*y2*) **by** *fastforce*
**show** *?thesis*
**proof**(*cases* (*x1*,*x2*) ∈ *fset xs* ∧ (*y1*,*y2*) ∈ *fset xs*)

**case** *0*: *True*
**then show** *?thesis*
**proof**(*cases e ∈ darcs x1 ∧ e ∈ darcs y1*)
**case** *True*
**then have** *1*: (*x1*,*x2*) = (*y1*,*y2*) ∨ ¬*wf-darcs* (*Node r xs*)
**using** *0 disjoint-darcs-if-wf-aux2* **by** *fast*
**then show** *?thesis* **using** *assms x-def y-def* **by** (*smt* (*z3*) *case-prod-conv comp-apply*)
**next**
**case** *False*
**then show** *?thesis*
**proof**(*cases x2=e*)
**case** *True*
**then show** *?thesis* **using** *assms x-def y-def disjoint-darcs-if-wf* **by** *force*
**next**
**case** *False*
**then show** *?thesis* **using** *assms x-def y-def disjoint-darcs-if-wf* **by** *fastforce*
**qed**
**qed**
**next**
**case** *False*
**then show** *?thesis* **using** *assms* **by** (*simp add*: *x-def y-def*)
**qed**
**qed**

**lemma** *dhead-commute*:
*comp-fun-commute* (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset xs* ∨ *e* ∉ (*darcs x* ∪ {*e2*}) ∨ ¬*wf-darcs* (*Node r xs*)
*then b else if e=e2 then root x else dhead x def e*)
**using** *dhead-commute-aux*[*of xs*] **by** *unfold-locales blast*

**lemma** *dhead-ffold-f-alt*:
**assumes** *P* = (λ*xs*. *wf-darcs* (*Node r xs*)) **and** *Q* = (λ(*x*,*e2*) -. *e* ∈ (*darcs x* ∪ {*e2*}))
**and** *R* = (λ(*x*,*e2*) -. *if e=e2 then root x else dhead x def e*)
**shows** (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset xs* ∨ *e* ∉ (*darcs x* ∪ {*e2*}) ∨ ¬*wf-darcs* (*Node r xs*) *then b*
*else if e=e2 then root x else dhead x def e*)
= (λ*a b*. *if a* ∉ *fset xs* ∨ ¬ *Q a b* ∨ ¬ *P xs then b else R a b*)
**using** *assms* **by** *fast*

**lemma** *dhead-ffold-f-alt-commute*:
**assumes** *P* = (λ*xs*. *wf-darcs* (*Node r xs*)) **and** *Q* = (λ(*x*,*e2*) -. *e* ∈ (*darcs x* ∪ {*e2*}))
**and** *R* = (λ(*x*,*e2*) -. *if e=e2 then root x else dhead x def e*)
**shows** *comp-fun-commute* (λ*a b*. *if a* ∉ *fset xs* ∨ ¬ *Q a b* ∨ ¬ *P xs then b else R a b*)
**using** *dhead-commute*[*of xs e r def*] *dhead-ffold-f-alt*[*OF assms*] **by** *simp*

**lemma** *dhead-ffold-supset*:
**assumes** *xs* $|\subseteq|$ *ys* **and** *wf-darcs* (*Node r ys*)
**shows** *ffold* ($\lambda$(*x*,*e2*) *b*. *if* (*x*,*e2*) $\notin$ *fset ys* $\vee$ *e* $\notin$ (*darcs x* $\cup$ {*e2*}) $\vee$ $\neg$*wf-darcs* (*Node r ys*) *then b*
*else if e*=*e2 then root x else dhead x def e*) (*def e*) *xs*
= *ffold* ($\lambda$(*x*,*e2*) *b*. *if* (*x*,*e2*) $\notin$ *fset xs* $\vee$ *e* $\notin$ (*darcs x* $\cup$ {*e2*}) $\vee$ $\neg$*wf-darcs* (*Node r xs*) *then b*
*else if e*=*e2 then root x else dhead x def e*) (*def e*) *xs*
(**is** *ffold ?f - - = ffold ?g - -*)
**proof** −
**let** *?P* = $\lambda$*xs*. *wf-darcs* (*Node r xs*)
**let** *?Q* = $\lambda$(*x*,*e2*) -. *e* $\in$ (*darcs x* $\cup$ {*e2*})
**let** *?R* = $\lambda$(*x*,*e2*) -. *if e*=*e2 then root x else dhead x def e*
**have** *0*: $\bigwedge$*xs*. *comp-fun-commute* ($\lambda$*a b*. *if a* $\notin$ *fset xs* $\vee$ $\neg$ *?Q a b* $\vee$ $\neg$ *?P xs then b else ?R a b*)
**using** *dhead-ffold-f-alt-commute* **by** *fast*
**have** *ffold* ($\lambda$*a b*. *if a* $\notin$ *fset ys* $\vee$ $\neg$ *?Q a b* $\vee$ $\neg$ *?P ys then b else ?R a b*) (*def e*) *xs*
= *ffold* ($\lambda$*a b*. *if a* $\notin$ *fset xs* $\vee$ $\neg$ *?Q a b* $\vee$ $\neg$ *?P xs then b else ?R a b*) (*def e*) *xs*
**using** *ffold-commute-supset*[*OF assms*(*1*), *of ?P ?Q ?R*, *OF assms*(*2*) *wf-darcs-sub 0*] **by** *simp*
**moreover have** *?f* = ($\lambda$*a b*. *if a* $\notin$ *fset ys* $\vee$ $\neg$ *?Q a b* $\vee$ $\neg$ *?P ys then b else ?R a b*) **by** *fast*
**moreover have** *?g* = ($\lambda$*a b*. *if a* $\notin$ *fset xs* $\vee$ $\neg$ *?Q a b* $\vee$ $\neg$ *?P xs then b else ?R a b*) **by** *fast*
**ultimately show** *?thesis* **by** *argo*
**qed**

**lemma** *dhead-in-child-eq-child-ffold*:
**assumes** (*t*,*e1*) $\in$ *fset xs* **and** *e* $\in$ *darcs t* **and** *wf-darcs* (*Node r xs*)
**shows** *ffold* ($\lambda$(*x*,*e2*) *b*. *if* (*x*,*e2*) $\notin$ *fset xs* $\vee$ *e* $\notin$ (*darcs x* $\cup$ {*e2*}) $\vee$ $\neg$*wf-darcs* (*Node r xs*)
*then b else if e*=*e2 then root x else dhead x def e*) (*def e*) *xs*
= *dhead t def e*
**using** *assms* **proof**(*induction xs*)
**case** (*insert x′ xs*)
**let** *?f* = ($\lambda$(*x*,*e2*) *b*. *if* (*x*,*e2*) $\notin$ *fset* (*finsert x′ xs*) $\vee$ *e* $\notin$ (*darcs x* $\cup$ {*e2*})
$\vee$ $\neg$*wf-darcs* (*Node r* (*finsert x′ xs*))
*then b else if e*=*e2 then root x else dhead x def e*)
**let** *?f′* = ($\lambda$(*x*,*e2*) *b*. *if* (*x*,*e2*) $\notin$ *fset xs* $\vee$ *e* $\notin$ (*darcs x* $\cup$ {*e2*}) $\vee$ $\neg$*wf-darcs* (*Node r xs*) *then b*
*else if e*=*e2 then root x else dhead x def e*)
**obtain** *x e3* **where** *x-def*: *x′* = (*x*,*e3*) **by** *fastforce*
**show** *?case*
**proof**(*cases x*=*t*)
**case** *True*
**have** *ffold ?f* (*def e*) (*finsert x′ xs*) = (*?f x′* (*ffold ?f* (*def e*) *xs*))
**using** *comp-fun-commute.ffold-finsert*[*of ?f x′ xs def e*] *dhead-commute in-*

*sert.hyps* **by** *fast*
  **also have** ... = (*?f* (*x*,*e3*) (*ffold ?f* (*def e*) *xs*)) **using** *x-def* **by** *blast*
  **also have** ... = *dhead x def e*
    **using** *x-def insert.prems*(*2*,*3*) *True disjoint-darcs-if-wf* **by** *fastforce*
  **finally show** *?thesis* **using** *True* **by** *blast*
 **next**
  **case** *False*
  **then have** *0*: (*t*,*e1*) ∈ *fset xs* **using** *insert.prems*(*1*) *x-def* **by** *simp*
 **have** *1*: *wf-darcs* (*Node r xs*) **using** *wf-darcs-sub*[*OF fsubset-finsertI insert.prems*(*3*)]
.
  **have** *2*: *xs* |⊆| (*finsert x' xs*) **by** *blast*
  **have** *3*: *e3* ≠ *e e* ∉ *darcs x*
    **using** *insert.prems*(*1*−*3*) *disjoint-darcs-if-wf x-def False* **by** *fastforce*+
  **have** *ffold ?f* (*def e*) (*finsert x' xs*) = (*?f x'* (*ffold ?f* (*def e*) *xs*))
     **using** *comp-fun-commute.ffold-finsert*[*of ?f x' xs def e*] *dhead-commute in-*
*sert.hyps* **by** *fast*
  **also have** ... = (*?f* (*x*,*e3*) (*ffold ?f* (*def e*) *xs*)) **using** *x-def* **by** *blast*
  **also have** ... = (*ffold ?f* (*def e*) *xs*) **using** *3* **by** *simp*
  **also have** ... = (*ffold ?f'* (*def e*) *xs*)
     **using** *dhead-ffold-supset*[*of xs finsert x' xs*] *insert.prems*(*3*) *2* **by** *simp*
  **also have** ... = *dhead t def e* **using** *insert.IH 0 1 insert.prems*(*2*) **by** *fast*
  **finally show** *?thesis* **.**
 **qed**
**qed**(*simp*)

**lemma** *dhead-in-child-eq-child*:
 **assumes** (*t*,*e1*) ∈ *fset xs* **and** *e* ∈ *darcs t* **and** *wf-darcs* (*Node r xs*)
  **shows** *dhead* (*Node r xs*) *def e* = *dhead t def e*
 **using** *assms dhead-in-child-eq-child-ffold*[*of t*] **by** *simp*

**lemma** *dhead-ffold-notelem-eq-def*:
 **assumes** ∀ (*t*,*e1*) ∈ *fset xs*. *e* ∉ *darcs t* ∧ *e* ≠ *e1*
 **shows** *ffold* (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset ys* ∨ *e* ∉ (*darcs x* ∪ {*e2*}) ∨ ¬*wf-darcs*
(*Node r ys*) *then b*
        *else if e*=*e2 then root x else dhead x def e*) (*def e*) *xs* = *def e*
**using** *assms* **proof**(*induction xs*)
 **case** *empty*
 **show** *?case*
  **apply** (*rule comp-fun-commute.ffold-empty*)
  **using** *dhead-commute* **by** *force*
**next**
 **case** (*insert x' xs*)
 **obtain** *x e3* **where** *x-def*: *x'* = (*x*,*e3*) **by** *fastforce*
 **let** *?f* = (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset ys* ∨ *e* ∉ (*darcs x* ∪ {*e2*}) ∨ ¬*wf-darcs*
(*Node r ys*)
        *then b else if e*=*e2 then root x else dhead x def e*)
 **have** *ffold ?f* (*def e*) (*finsert x' xs*) = *?f x'* (*ffold ?f* (*def e*) *xs*)
   **using** *comp-fun-commute.ffold-finsert*[*of ?f x' xs*] *dhead-commute insert.hyps*
**by** *fast*

**also have** ... = (*ffold ?f* (*def e*) *xs*) **using** *insert.prems* **by** *auto*
**also have** ... = *def e* **using** *insert.IH insert.prems* **by** *simp*
**finally show** *?case* **.**
**qed**

**lemma** *dhead-notelem-eq-def*:
**assumes** *e* ∉ *darcs t*
**shows** *dhead t def e* = *def e*
**proof** −
**obtain** *r xs* **where** *xs-def*[*simp*]: *t* = *Node r xs* **using** *dtree.exhaust* **by** *auto*
**let** *?f* = (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset xs* ∨ *e* ∉ (*darcs x* ∪ {*e2*}) ∨ ¬*wf-darcs* (*Node r xs*)
*then b else if e*=*e2 then root x else dhead x def e*)
**have** *0*: ∀(*t*, *e1*)∈*fset xs*. *e* ∉ *darcs t* ∧ *e1*≠*e* **using** *assms* **by** *auto*
**have** *dhead* (*Node r xs*) *def e* = *ffold ?f* (*def e*) *xs* **by** *simp*
**then show** *?thesis* **using** *dhead-ffold-notelem-eq-def 0* **by** *fastforce*
**qed**

**lemma** *dhead-in-set-eq-root-ffold*:
**assumes** (*t*,*e*) ∈ *fset xs* **and** *wf-darcs* (*Node r xs*)
**shows** *ffold* (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset xs* ∨ *e* ∉ (*darcs x* ∪ {*e2*}) ∨ ¬*wf-darcs* (*Node r xs*)
*then b else if e*=*e2 then root x else dhead x def e*) (*def e*) *xs*
= *root t* (**is** *ffold ?f' - - = -*)
**using** *assms* **proof**(*induction xs*)
**case** (*insert x' xs*)
**let** *?f* = (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset* (*finsert x' xs*) ∨ *e* ∉ (*darcs x* ∪ {*e2*})
∨ ¬*wf-darcs* (*Node r* (*finsert x' xs*))
*then b else if e*=*e2 then root x else dhead x def e*)
**let** *?f'* = (λ(*x*,*e2*) *b*. *if* (*x*,*e2*) ∉ *fset xs* ∨ *e* ∉ (*darcs x* ∪ {*e2*}) ∨ ¬*wf-darcs* (*Node r xs*) *then b*
*else if e*=*e2 then root x else dhead x def e*)
**obtain** *x e3* **where** *x-def*: *x'* = (*x*,*e3*) **by** *fastforce*
**show** *?case*
**proof**(*cases e3*=*e*)
**case** *True*
**then have** *x*=*t* **using** *insert.prems*(*1*,*2*) *x-def disjoint-darcs-if-wf* **by** *fastforce*
**have** *ffold ?f* (*def e*) (*finsert x' xs*) = (*?f x'* (*ffold ?f* (*def e*) *xs*))
**using** *comp-fun-commute.ffold-finsert*[*of ?f x' xs def e*] *dhead-commute insert.hyps* **by** *fast*
**also have** ... = (*?f* (*x*,*e3*) (*ffold ?f* (*def e*) *xs*)) **using** *x-def* **by** *blast*
**also have** ... = *root x* **using** *x-def insert.prems*(*1*,*2*) *True* **by** *simp*
**finally show** *?thesis* **using** *True* ‹*x*=*t*› **by** *blast*
**next**
**case** *False*
**then have** *0*: (*t*,*e*) ∈ *fset xs* **using** *insert.prems*(*1*) *x-def* **by** *simp*
**have** *1*: *wf-darcs* (*Node r xs*) **using** *wf-darcs-sub*[*OF fsubset-finsertI insert.prems*(*2*)]
.
**have** *2*: *xs* |⊆| (*finsert x' xs*) **by** *blast*

**have** *3*: *e3* ≠ *e* **using** *insert.prems*(*2*) *False* **by** *simp*
**have** *4*: *e* ∉ (*darcs x* ∪ {*e3*})
**using** *insert.prems*(*1*−*2*) *False x-def disjoint-darcs-if-wf* **by** *fastforce*
**have** *ffold ?f* (*def e*) (*finsert x′ xs*) = (*?f x′* (*ffold ?f* (*def e*) *xs*))
**using** *comp-fun-commute.ffold-finsert*[*of ?f x′ xs def e*] *dhead-commute insert.hyps* **by** *fast*
**also have** ... = (*?f* (*x*,*e3*) (*ffold ?f* (*def e*) *xs*)) **using** *x-def* **by** *blast*
**also have** ... = (*ffold ?f* (*def e*) *xs*) **using** *4* **by** *auto*
**also have** ... = (*ffold ?f′* (*def e*) *xs*)
**using** *dhead-ffold-supset*[*of xs finsert x′ xs*] *insert.prems*(*2*) *2* **by** *simp*
**also have** ... = *root t* **using** *insert.IH 0 1 insert.prems*(*2*) **by** *blast*
**finally show** *?thesis* **.**
**qed**
**qed**(*simp*)

**lemma** *dhead-in-set-eq-root*:
⟦(*t*,*e*) ∈ *fset xs*; *wf-darcs* (*Node r xs*)⟧ ⟹ *dhead* (*Node r xs*) *def e* = *root t*
**using** *dhead-in-set-eq-root-ffold*[*of t*] **by** *simp*

**lemma** *self-subtree*: *is-subtree t t*
**using** *is-subtree.elims*(*3*) **by** *blast*

**lemma** *subtree-trans*: *is-subtree x y* ⟹ *is-subtree y z* ⟹ *is-subtree x z*
**by** (*induction z*) *fastforce*+

**lemma** *subtree-trans′*: *transp is-subtree*
**using** *subtree-trans transpI* **by** *auto*

**lemma** *subtree-if-child*: *x* ∈ *fst* ‘ *fset xs* ⟹ *is-subtree x* (*Node r xs*)
**using** *is-subtree.elims*(*3*) **by** *force*

**lemma** *subtree-if-suc*: *t1* ∈ *fst* ‘ *fset* (*sucs t2*) ⟹ *is-subtree t1 t2*
**using** *subtree-if-child*[*of t1 sucs t2 root t2*] **by** *simp*

**lemma** *child-sub-if-strict-subtree*:
⟦*strict-subtree t1* (*Node r xs*)⟧ ⟹ ∃ *t3* ∈ *fst* ‘ *fset xs*. *is-subtree t1 t3*
**unfolding** *strict-subtree-def* **by** *force*

**lemma** *suc-sub-if-strict-subtree*:
*strict-subtree t1 t2* ⟹ ∃ *t3* ∈ *fst* ‘ *fset* (*sucs t2*). *is-subtree t1 t3*
**using** *child-sub-if-strict-subtree*[*of t1 root t2*] **by** *simp*

**lemma** *subtree-size-decr*: ⟦*is-subtree t1 t2*; *t1* ≠ *t2*⟧ ⟹ *size t1* < *size t2*
**using** *dtree-size-decr-aux* **by**(*induction t2*) *fastforce*

**lemma** *subtree-size-decr′*: *strict-subtree t1 t2* ⟹ *size t1* < *size t2*
**unfolding** *strict-subtree-def* **using** *dtree-size-decr-aux* **by**(*induction t2*) *fastforce*

**lemma** *subtree-size-le*: *is-subtree t1 t2* ⟹ *size t1* ≤ *size t2*

**using** *subtree-size-decr* **by** *fastforce*

**lemma** *subtree-antisym*: ⟦*is-subtree t1 t2*; *is-subtree t2 t1*⟧ ⟹ *t1* = *t2*
**using** *subtree-size-le subtree-size-decr* **by** *fastforce*

**lemma** *subtree-antisym'*: *antisymp is-subtree*
**using** *antisympI subtree-antisym* **by** *blast*

**corollary** *subtree-eq-if-trans-eq1*: ⟦*is-subtree t1 t2*; *is-subtree t2 t3*; *t1* = *t3*⟧ ⟹ *t1* = *t2*
**using** *subtree-antisym* **by** *blast*

**corollary** *subtree-eq-if-trans-eq2*: ⟦*is-subtree t1 t2*; *is-subtree t2 t3*; *t1* = *t3*⟧ ⟹ *t2* = *t3*
**using** *subtree-antisym* **by** *blast*

**lemma** *subtree-partial-ord*: *class.order is-subtree strict-subtree*
**by** *standard* (*auto simp*: *self-subtree subtree-antisym strict-subtree-def intro*: *subtree-trans*)

**lemma** *finite-subtrees*: *finite* {*x*. *is-subtree x t*}
**by** (*induction t*) *auto*

**lemma** *subtrees-insert-union*:
{*x*. *is-subtree x* (*Node r xs*)} = *insert* (*Node r xs*) (⋃ *t1* ∈ *fst* ‘ *fset xs*. {*x*. *is-subtree x t1*})
**by** *fastforce*

**lemma** *subtrees-insert-union-suc*:
{*x*. *is-subtree x t*} = *insert t* (⋃ *t1* ∈ *fst* ‘ *fset* (*sucs t*). {*x*. *is-subtree x t1*})
**using** *subtrees-insert-union*[*of root t sucs t*] **by** *simp*

**lemma** *darcs-subtree-subset*: *is-subtree x y* ⟹ *darcs x* ⊆ *darcs y*
**by**(*induction y*) *force*

**lemma** *dverts-subtree-subset*: *is-subtree x y* ⟹ *dverts x* ⊆ *dverts y*
**by**(*induction y*) *force*

**lemma** *single-subtree-root-dverts*:
*is-subtree* (*Node v2* {|(*t2*, *e2*)|}) *t1* ⟹ *v2* ∈ *dverts t1*
**by** (*fastforce dest*: *dverts-subtree-subset*)

**lemma** *single-subtree-child-root-dverts*:
*is-subtree* (*Node v2* {|(*t2*, *e2*)|}) *t1* ⟹ *root t2* ∈ *dverts t1*
**by** (*fastforce simp*: *dtree.set-sel*(*1*) *dest*: *dverts-subtree-subset*)

**lemma** *subtree-root-if-dverts*: *x* ∈ *dverts t* ⟹ ∃ *xs*. *is-subtree* (*Node x xs*) *t*
**by**(*induction t*) *fastforce*

**lemma** *subtree-child-if-strict-subtree*:
*strict-subtree t1 t2* $\Longrightarrow \exists r\ xs.$ *is-subtree* (*Node r xs*) *t2* $\wedge$ *t1* $\in$ *fst* ‘ *fset xs*
**proof**(*induction t2*)
**case** (*Node r xs*)
**then obtain** *t e* **where** *t-def*: (*t*,*e*) $\in$ *fset xs is-subtree t1 t*
**unfolding** *strict-subtree-def* **by** *auto*
**show** *?case*
**proof**(*cases t1* = *t*)
**case** *True*
**then show** *?thesis* **using** *t-def* **by** *force*
**next**
**case** *False*
**then show** *?thesis* **using** *Node.IH*[*OF t-def*(*1*)] *t-def* **unfolding** *strict-subtree-def*
**by** *auto*
**qed**
**qed**

**lemma** *subtree-child-if-dvert-notroot*:
**assumes** *v* $\neq$ *r* **and** *v* $\in$ *dverts* (*Node r xs*)
**shows** $\exists r'\ ys\ zs.$ *is-subtree* (*Node r' ys*) (*Node r xs*) $\wedge$ *Node v zs* $\in$ *fst* ‘ *fset ys*
**proof** −
**obtain** *zs* **where** *sub*: *is-subtree* (*Node v zs*) (*Node r xs*)
**using** *assms*(*2*) *subtree-root-if-dverts* **by** *fast*
**then show** *?thesis* **using** *subtree-child-if-strict-subtree strict-subtree-def assms*(*1*)
**by** *fast*
**qed**

**lemma** *subtree-child-if-dvert-notelem*:
⟦*v* $\neq$ *root t*; *v* $\in$ *dverts t*⟧ $\Longrightarrow \exists r'\ ys\ zs.$ *is-subtree* (*Node r' ys*) *t* $\wedge$ *Node v zs* $\in$ *fst* ‘ *fset ys*
**using** *subtree-child-if-dvert-notroot*[*of v root t sucs t*] **by** *simp*

**lemma** *strict-subtree-subset*:
**assumes** *strict-subtree t* (*Node r xs*) **and** *xs* |⊆| *ys*
**shows** *strict-subtree t* (*Node r ys*)
**proof** −
**obtain** *t1 e1* **where** *t1-def*: (*t1*,*e1*) $\in$ *fset xs is-subtree t t1*
**using** *assms*(*1*) **unfolding** *strict-subtree-def* **by** *auto*
**have** *size t* < *size* (*Node r xs*) **using** *subtree-size-decr'*[*OF assms*(*1*)] **by** *blast*
**then have** *size t* < *size* (*Node r ys*) **using** *size-le-if-child-subset*[*OF assms*(*2*)]
**by** *simp*
**moreover have** *is-subtree t* (*Node r ys*) **using** *assms*(*2*) *t1-def* **by** *auto*
**ultimately show** *?thesis* **unfolding** *strict-subtree-def* **by** *blast*
**qed**

**lemma** *strict-subtree-singleton*:
⟦*strict-subtree t* (*Node r* {|*x*|}); *x* |∈| *xs*⟧
$\Longrightarrow$ *strict-subtree t* (*Node r xs*)
**using** *strict-subtree-subset* **by** *fast*

### 7.3.1 Finite Directed Trees to Dtree

**context** *finite-directed-tree*
**begin**

**lemma** *child-subtree*:
  **assumes** $e \in$ *out-arcs* $T\ r$
  **shows** $\{x.\ (head\ T\ e) \rightarrow^*_T x\} \subseteq \{x.\ r \rightarrow^*_T x\}$
**proof** −
  **have** $r \rightarrow^*_T (head\ T\ e)$ **using** *assms in-arcs-imp-in-arcs-ends* **by** *auto*
  **then show** *?thesis* **by** (*metis Collect-mono reachable-trans*)
**qed**

**lemma** *child-strict-subtree*:
  **assumes** $e \in$ *out-arcs* $T\ r$
  **shows** $\{x.\ (head\ T\ e) \rightarrow^*_T x\} \subset \{x.\ r \rightarrow^*_T x\}$
**proof** −
  **have** $r \rightarrow_T (head\ T\ e)$ **using** *assms in-arcs-imp-in-arcs-ends* **by** *auto*
  **then have** $\neg\ ((head\ T\ e) \rightarrow^*_T r)$ **using** *reachable1-not-reverse* **by** *blast*
  **then show** *?thesis* **using** *child-subtree assms* **by** *auto*
**qed**

**lemma** *child-card-decr*:
  **assumes** $e \in$ *out-arcs* $T\ r$
  **shows** *Finite-Set.card* $\{x.\ (head\ T\ e) \rightarrow^*_T x\} <$ *Finite-Set.card* $\{x.\ r \rightarrow^*_T x\}$
 **using** *assms child-strict-subtree* **by** (*meson psubset-card-mono reachable-verts-finite*)

**function** *to-dtree-aux* :: $'a \Rightarrow ('a,'b)$ *dtree* **where**
  *to-dtree-aux r* = *Node r* (*Abs-fset* $\{(x,e).$
    (*if e* $\in$ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)$\}$)
  **by** *auto*
**termination**
 **by**(*relation measure* ($\lambda r.$ *Finite-Set.card* $\{x.\ r \rightarrow^*_T x\}$)) (*auto simp*: *child-card-decr*)

**definition** *to-dtree* :: $('a,'b)$ *dtree* **where**
  *to-dtree* = *to-dtree-aux root*

**abbreviation** *from-dtree* :: $('a,'b)$ *dtree* $\Rightarrow$ $('a,'b)$ *pre-digraph* **where**
  *from-dtree t* ≡ *Dtree.from-dtree* (*tail T*) (*head T*) *t*

**lemma** *to-dtree-root-eq-root*[*simp*]: *Dtree.root to-dtree* = *root*
  **unfolding** *to-dtree-def* **by** *simp*

**lemma** *verts-fset-id*: *fset* (*Abs-fset* (*verts T*)) = *verts T*
  **by** (*simp add*: *Abs-fset-inverse*)

**lemma** *arcs-fset-id*: *fset* (*Abs-fset* (*arcs T*)) = *arcs T*
  **by** (*simp add*: *Abs-fset-inverse*)

**lemma** *dtree-leaf-child-empty*:

*leaf r* $\Longrightarrow$ {(*x*,*e*). (*if e* $\in$ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)} = {}
**unfolding** *leaf-def* **by** *simp*

**lemma** *dtree-leaf-no-children*: *leaf r* $\Longrightarrow$ *to-dtree-aux r* = *Node r* {||}
**using** *dtree-leaf-child-empty* **by** (*simp add*: *bot-fset.abs-eq*)

**lemma** *dtree-children-alt*:
{(*x*,*e*). (*if e* $\in$ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)}
= {(*x*,*e*). *e* $\in$ *out-arcs T r* $\wedge$ *x* = *to-dtree-aux* (*head T e*)}
**by** *metis*

**lemma** *dtree-children-img-alt*:
($\lambda$*e*. (*to-dtree-aux* (*head T e*),*e*)) ‘ (*out-arcs T r*)
= {(*x*,*e*). (*if e* $\in$ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)}
**using** *dtree-children-alt* **by** *blast*

**lemma** *dtree-children-fin*:
*finite* {(*x*,*e*). (*if e* $\in$ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)}
**using** *finite-imageI*[*of out-arcs T r* ($\lambda$*e*. (*to-dtree-aux* (*head T e*),*e*))]
*dtree-children-img-alt finite-out-arcs* **by** *fastforce*

**lemma** *dtree-children-fset-id*:
**assumes** *to-dtree-aux r* = *Node r xs*
**shows** *fset xs* = {(*x*,*e*). (*if e* $\in$ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)}
**proof** −
**let** *?xs* = {(*x*,*e*). (*if e* $\in$ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)}
**have** *finite ?xs* **using** *dtree-children-fin* **by** *simp*
**then have** *fset* (*Abs-fset ?xs*) = *?xs* **using** *Abs-fset-inverse* **by** *blast*
**then show** *?thesis* **using** *assms Abs-fset-inverse* **by** *simp*
**qed**

**lemma** *to-dtree-aux-empty-if-notT*:
**assumes** *r* $\notin$ *verts T*
**shows** *to-dtree-aux r* = *Node r* {||}
**proof**(*rule ccontr*)
**assume** *asm*: *to-dtree-aux r* $\neq$ *Node r* {||}
**then obtain** *xs* **where** *xs-def*: *Node r xs* = *to-dtree-aux r* **by** *simp*
**then have** *xs* $\neq$ {||} **using** *asm* **by** *simp*
**then obtain** *x e* **where** *x-def*: (*x*,*e*) $\in$ *fset xs* **by** *fast*
**then have** *e* $\in$ *out-arcs T r* **using** *xs-def dtree-children-fset-id*[*of r*] **by** (*auto split*: *if-splits*)
**then show** *False* **using** *assms* **by** *auto*
**qed**

**lemma** *to-dtree-aux-root*: *Dtree.root* (*to-dtree-aux r*) = *r*
**by** *simp*

**lemma** *out-arc-if-child*:
**assumes** $x \in (fst\ `\ \{(x,e).\ (if\ e \in out\text{-}arcs\ T\ r\ then\ x = to\text{-}dtree\text{-}aux\ (head\ T\ e)\ else\ False)\})$
**shows** $\exists\, e.\ e \in out\text{-}arcs\ T\ r \wedge x = to\text{-}dtree\text{-}aux\ (head\ T\ e)$
**proof** −
**let** *?xs* $= \{(x,e).\ (if\ e \in out\text{-}arcs\ T\ r\ then\ x = to\text{-}dtree\text{-}aux\ (head\ T\ e)\ else\ False)\}$
**have** $\exists\, y.\ y \in ?xs \wedge fst\ y = x$ **using** *assms* **by** *blast*
**then show** *?thesis* **by** (*smt* (*verit*, *best*) *case-prodE fst-conv mem-Collect-eq*)
**qed**

**lemma** *dominated-if-child-aux*:
**assumes** $x \in (fst\ `\ \{(x,e).\ (if\ e \in out\text{-}arcs\ T\ r\ then\ x = to\text{-}dtree\text{-}aux\ (head\ T\ e)\ else\ False)\})$
**shows** $r \rightarrow_T (Dtree.root\ x)$
**proof** −
**obtain** *e* **where** $e \in out\text{-}arcs\ T\ r \wedge x = to\text{-}dtree\text{-}aux\ (head\ T\ e)$
**using** *assms out-arc-if-child* **by** *blast*
**then show** *?thesis* **using** *in-arcs-imp-in-arcs-ends* **by** *force*
**qed**

**lemma** *dominated-if-child*:
$[\![to\text{-}dtree\text{-}aux\ r = Node\ r\ xs;\ x \in fst\ `\ fset\ xs]\!] \Longrightarrow r \rightarrow_T (Dtree.root\ x)$
**using** *dominated-if-child-aux dtree-children-fset-id* **by** *simp*

**lemma** *image-add-snd-snd-id*: $snd\ `\ ((\lambda e.\ (to\text{-}dtree\text{-}aux\ (head\ T\ e),e))\ `\ x) = x$
**by** (*intro equalityI subsetI*) (*force simp*: *image-iff*)+

**lemma** *to-dtree-aux-child-in-verts*:
**assumes** $Node\ r'\ xs = to\text{-}dtree\text{-}aux\ r$ **and** $x \in fst\ `\ fset\ xs$
**shows** $Dtree.root\ x \in verts\ T$
**proof** −
**have** $r \rightarrow_T Dtree.root\ x$ **using** *assms dominated-if-child* **by** *auto*
**then show** *?thesis* **using** *adj-in-verts*(*2*) **by** *auto*
**qed**

**lemma** *to-dtree-aux-parent-in-verts*:
**assumes** $Node\ r'\ xs = to\text{-}dtree\text{-}aux\ r$ **and** $x \in fst\ `\ fset\ xs$
**shows** $r \in verts\ T$
**proof** −
**have** $r \rightarrow_T Dtree.root\ x$ **using** *assms dominated-if-child* **by** *auto*
**then show** *?thesis* **using** *adj-in-verts*(*2*) **by** *auto*
**qed**

**lemma** *dtree-out-arcs*:
$snd\ `\ \{(x,e).\ (if\ e \in out\text{-}arcs\ T\ r\ then\ x = to\text{-}dtree\text{-}aux\ (head\ T\ e)\ else\ False)\} = out\text{-}arcs\ T\ r$
**using** *dtree-children-img-alt* **by** (*metis image-add-snd-snd-id*)

**lemma** *dtree-out-arcs-eq-snd*:

**assumes** *to-dtree-aux r = Node r xs*
**shows** *(snd ‘ (fset xs)) = out-arcs T r*
**using** *assms dtree-out-arcs dtree-children-fset-id* **by** *blast*

**lemma** *dtree-aux-fst-head-snd-aux*:
**assumes** *x ∈ {(x,e). (if e ∈ out-arcs T r then x = to-dtree-aux (head T e) else False)}*
**shows** *Dtree.root (fst x) = (head T (snd x))*
**using** *assms* **by** *(metis (mono-tags, lifting) Collect-case-prodD to-dtree-aux-root)*

**lemma** *dtree-aux-fst-head-snd*:
**assumes** *to-dtree-aux r = Node r xs* **and** *x ∈ fset xs*
**shows** *Dtree.root (fst x) = (head T (snd x))*
**using** *assms dtree-children-fset-id dtree-aux-fst-head-snd-aux* **by** *simp*

**lemma** *child-if-dominated-aux*:
**assumes** *r →T x*
**shows** *∃ y ∈ (fst ‘ {(x,e). (if e ∈ out-arcs T r then x = to-dtree-aux (head T e) else False)}).*
*Dtree.root y = x*
**proof** −
**let** *?xs = {(x,e). (if e ∈ out-arcs T r then x = to-dtree-aux (head T e) else False)}*
**obtain** *e* **where** *e-def*: *e ∈ out-arcs T r ∧ head T e = x* **using** *assms* **by** *auto*
**then have** *e ∈ snd ‘ ?xs* **using** *dtree-out-arcs* **by** *auto*
**then obtain** *y* **where** *y-def*: *y ∈ ?xs ∧ snd y = e* **by** *blast*
**then have** *Dtree.root (fst y) = head T e* **using** *dtree-aux-fst-head-snd-aux* **by** *blast*
**then show** *?thesis* **using** *e-def y-def* **by** *blast*
**qed**

**lemma** *child-if-dominated*:
**assumes** *to-dtree-aux r = Node r xs* **and** *r →T x*
**shows** *∃ y ∈ (fst ‘ (fset xs)). Dtree.root y = x*
**using** *assms child-if-dominated-aux dtree-children-fset-id* **by** *presburger*

**lemma** *to-dtree-aux-reach-in-dverts*: *⟦t = to-dtree-aux r; r →*T x⟧ ⟹ x ∈ dverts t*
**proof**(*induction t arbitrary*: *r rule*: *darcs-mset.induct*)
**case** *(1 r′ xs)*
**then have** *r = r′* **by** *simp*
**then show** *?case*
**proof**(*cases r=x*)
**case** *True*
**then show** *?thesis* **using** ‹*r = r′*› **by** *simp*
**next**
**case** *False*
**then have** *r →*+T x* **using** *1.prems(2)* **by** *blast*
**then have** *∃ r′. r →T r′ ∧ r′ →*T x*
**by** *(metis False converse-reachable-cases reachable1-reachable)*

```
    then obtain x' where x'-def: r →_T x' ∧ x' →*_T x by blast
    then obtain y where y-def: y ∈ fst ` fset xs ∧ Dtree.root y = x'
      using 1.prems(1) child-if-dominated by fastforce
    then obtain yp where yp-def: fst yp = y ∧ yp ∈ fset xs using y-def by blast
    from y-def have y = to-dtree-aux x'
      using 1.prems(1) dtree-children-fset-id ‹r=r'›
      by (metis (no-types, lifting) out-arc-if-child to-dtree-aux-root)
    then have x ∈ dverts y using 1.IH prod.exhaust-sel yp-def x'-def by metis
    then show ?thesis using dtree.set-intros(2) y-def by auto
  qed
qed

lemma to-dtree-aux-dverts-reachable:
  ⟦t = to-dtree-aux r; x ∈ dverts t; r ∈ verts T⟧ ⟹ r →*_T x
proof(induction t arbitrary: r rule: darcs-mset.induct)
  case (1 r' xs)
  then have r = r' by simp
  then show ?case
  proof(cases r=x)
    case True
    then show ?thesis using 1.prems(3) by auto
  next
    case False
    then obtain y where y-def: y ∈ fst ` fset xs ∧ x ∈ dverts y
      using 1.prems(2) ‹r = r'› by fastforce
    then have 0: r →_T Dtree.root y using 1.prems(1) ‹r = r'› dominated-if-child
by simp
    then have 2: Dtree.root y ∈ verts T using adj-in-verts(2) by auto
    obtain yp where yp-def: fst yp = y ∧ yp ∈ fset xs using y-def by blast
    have ∃ yr. y = to-dtree-aux yr
      using 1.prems(1) y-def dtree-children-fset-id
      by (metis (no-types, lifting) ‹r = r'› out-arc-if-child)
    then have Dtree.root y →*_T x
      using 1.IH 2 y-def yp-def surjective-pairing to-dtree-aux-root by metis
    then show ?thesis using 0 adj-reachable-trans by auto
  qed
qed

lemma dverts-eq-reachable: r ∈ verts T ⟹ dverts (to-dtree-aux r) = {x. r →*_T
x}
  using to-dtree-aux-reach-in-dverts to-dtree-aux-dverts-reachable by blast

lemma dverts-eq-reachable': ⟦r ∈ verts T; t = to-dtree-aux r⟧ ⟹ dverts t = {x.
r →*_T x}
  using dverts-eq-reachable by blast

lemma dverts-eq-verts: dverts to-dtree = verts T
 unfolding to-dtree-def using dverts-eq-reachable reachable-from-root reachable-in-verts(2)
 by (metis mem-Collect-eq root-in-T subsetI subset-antisym)
```

**lemma** *arc-out-arc*: $e \in arcs\ T \Longrightarrow \exists\, v \in verts\ T.\ e \in out\text{-}arcs\ T\ v$
**by** *simp*

**lemma** *darcs-in-out-arcs*: $t = to\text{-}dtree\text{-}aux\ r \Longrightarrow e \in darcs\ t \Longrightarrow \exists\, v{\in}dverts\ t.\ e \in out\text{-}arcs\ T\ v$
**proof**(*induction t arbitrary*: *r rule*: *darcs-mset.induct*)
**case** (*1 r' xs*)
**then show** *?case*
**proof**(*cases e* ∈ *snd* ‘ *fset xs*)
**case** *True*
**then show** *?thesis*
**using** *1.prems*(*1*) *dtree-out-arcs-eq-snd to-dtree-aux-root*
**by** (*metis dtree.set-intros*(*1*) *dtree.sel*(*1*))
**next**
**case** *False*
**then have** ∃ *y* ∈ *fst* ‘ *fset xs*. *e* ∈ *darcs y* **using** *1.prems*(*2*) **by** *force*
**then obtain** *y* **where** *y-def*: *y* ∈ *fst* ‘ *fset xs* ∧ *e* ∈ *darcs y* **by** *blast*
**obtain** *yp* **where** *yp-def*: *fst yp* = *y* ∧ *yp* ∈ *fset xs* **using** *y-def* **by** *blast*
**have** *0*: (*y*, *snd yp*) = *yp* **using** *yp-def* **by** *auto*
**have** ∃ *yr*. *y* = *to-dtree-aux yr*
**using** *1.prems*(*1*) *y-def dtree-children-fset-id*
**by** (*metis* (*no-types*, *lifting*) *dtree.sel*(*1*) *out-arc-if-child to-dtree-aux-root*)
**then have** ∃ *v*∈*dverts y*. *e* ∈ *out-arcs T v* **using** *1.IH 0 y-def yp-def* **by** *blast*
**then obtain** *v* **where** *v* ∈ *dverts y* ∧ *e* ∈ *out-arcs T v* **by** *blast*
**then show** *?thesis* **using** *y-def* **by** *auto*
**qed**
**qed**

**lemma** *darcs-in-arcs*: $e \in darcs\ to\text{-}dtree \Longrightarrow e \in arcs\ T$
**using** *darcs-in-out-arcs out-arcs-in-arcs to-dtree-def* **by** *fast*

**lemma** *out-arcs-in-darcs*: $t = to\text{-}dtree\text{-}aux\ r \Longrightarrow \exists\, v{\in}dverts\ t.\ e \in out\text{-}arcs\ T\ v \Longrightarrow e \in darcs\ t$
**proof**(*induction t arbitrary*: *r rule*: *darcs-mset.induct*)
**case** (*1 r' xs*)
**then have** *r'* = *r* **by** *simp*
**then obtain** *v* **where** *v-def*: *v*∈*dverts* (*Node r xs*) ∧ *e* ∈ *out-arcs T v* **using** *1.prems*(*2*) **by** *blast*
**then show** *?case*
**proof**(*cases e* ∈ *snd* ‘ *fset xs*)
**case** *True*
**then show** *?thesis* **by** *force*
**next**
**case** *False*
**then have** *e* ∉ *out-arcs T r* **using** *1.prems*(*1*) ‹*r'* = *r*› *dtree-out-arcs-eq-snd* **by** *metis*
**then have** *v* ≠ *r* **using** *v-def* **by** *blast*
**then obtain** *y* **where** *y-def*: *y* ∈ *fst* ‘ *fset xs* ∧ *v* ∈ *dverts y* **using** *v-def* **by**

*force*
**then obtain** *yp* **where** *yp-def*: *fst yp* = *y* ∧ *yp* ∈ *fset xs* **by** *blast*
**have** *0*: (*y*, *snd yp*) = *yp* **using** *yp-def* **by** *auto*
**have** ∃ *yr*. *y* = *to-dtree-aux yr*
**using** *1.prems*(*1*) *y-def dtree-children-fset-id*
**by** (*metis* (*no-types*, *lifting*) *dtree.sel*(*1*) *out-arc-if-child to-dtree-aux-root*)
**then have** *e* ∈ *darcs y* **using** *1.IH 0 v-def y-def yp-def* **by** *blast*
**then show** *?thesis* **using** *y-def* **by** *force*
**qed**
**qed**

**lemma** *arcs-in-darcs*: *e* ∈ *arcs T* ⟹ *e* ∈ *darcs to-dtree*
**using** *arc-out-arc out-arcs-in-darcs dverts-eq-verts to-dtree-def* **by** *fast*

**lemma** *darcs-eq-arcs*: *darcs to-dtree* = *arcs T*
**using** *arcs-in-darcs darcs-in-arcs* **by** *blast*

**lemma** *to-dtree-aux-self*:
**assumes** *Node r xs* = *to-dtree-aux r* **and** (*y*,*e*) ∈ *fset xs*
**shows** *y* = *to-dtree-aux* (*Dtree.root y*)
**proof** −
**have** ∃ *y′*. *y* = *to-dtree-aux y′*
**using** *assms dtree-children-fset-id* **by** (*metis* (*mono-tags*, *lifting*) *case-prodD mem-Collect-eq*)
**then obtain** *y′* **where** *y* = *to-dtree-aux y′* **by** *blast*
**then show** *?thesis* **by** *simp*
**qed**

**lemma** *to-dtree-aux-self-subtree*:
⟦*t1* = *to-dtree-aux r*; *is-subtree t2 t1*⟧ ⟹ *t2* = *to-dtree-aux* (*Dtree.root t2*)
**proof**(*induction t1 arbitrary*: *r*)
**case** (*Node r′ xs*)
**then show** *?case*
**proof**(*cases Node r′ xs* = *t2*)
**case** *True*
**then show** *?thesis* **using** *Node.prems*(*1*) **by** *force*
**next**
**case** *False*
**then obtain** *t e* **where** *t-def*: (*t*,*e*) ∈ *fset xs is-subtree t2 t* **using** *Node.prems*(*2*) **by** *auto*
**then have** *t* = *to-dtree-aux* (*Dtree.root t*) **using** *Node.prems*(*1*) *to-dtree-aux-self* **by** *simp*
**then show** *?thesis* **using** *Node.IH*[*of* (*t*,*e*) *t Dtree.root t*] *t-def* **by** *simp*
**qed**
**qed**

**lemma** *to-dtree-self-subtree*: *is-subtree t to-dtree* ⟹ *t* = *to-dtree-aux* (*Dtree.root t*)
**unfolding** *to-dtree-def* **using** *to-dtree-aux-self-subtree* **by** *blast*

**lemma** *to-dtree-self-subtree'*: *is-subtree* (*Node r xs*) *to-dtree* $\Longrightarrow$ (*Node r xs*) = *to-dtree-aux r*
**using** *to-dtree-self-subtree*[*of Node r xs*] **by** *simp*

**lemma** *child-if-dominated-to-dtree*:
⟦*is-subtree* (*Node r xs*) *to-dtree*; *r* $\rightarrow_T$ *v*⟧ $\Longrightarrow$ $\exists$ *t*. *t* ∈ *fst* ' *fset xs* ∧ *Dtree.root t* = *v*
**using** *child-if-dominated*[*of r*] *to-dtree-self-subtree'* **by** *simp*

**lemma** *child-if-dominated-to-dtree'*:
⟦*is-subtree* (*Node r xs*) *to-dtree*; *r* $\rightarrow_T$ *v*⟧ $\Longrightarrow$ $\exists$ *ys*. *Node v ys* ∈ *fst* ' *fset xs*
**using** *child-if-dominated-to-dtree dtree.exhaust dtree.sel*(*1*) **by** *metis*

**lemma** *child-darc-tail-parent*:
**assumes** *Node r xs* = *to-dtree-aux r* **and** (*x*,*e*) ∈ *fset xs*
**shows** *tail T e* = *r*
**proof** −
**have** *e* ∈ *out-arcs T r*
**using** *assms dtree-children-fset-id* **by** (*metis* (*no-types*, *lifting*) *case-prodD mem-Collect-eq*)
**then show** *?thesis* **by** *simp*
**qed**

**lemma** *child-darc-head-root*:
⟦*Node r xs* = *to-dtree-aux r*; (*t*,*e*) ∈ *fset xs*⟧ $\Longrightarrow$ *head T e* = *Dtree.root t*
**using** *dtree-aux-fst-head-snd* **by** *force*

**lemma** *child-darc-in-arcs*:
**assumes** *Node r xs* = *to-dtree-aux r* **and** (*x*,*e*) ∈ *fset xs*
**shows** *e* ∈ *arcs T*
**proof** −
**have** *e* ∈ *out-arcs T r*
**using** *assms dtree-children-fset-id* **by** (*metis* (*no-types*, *lifting*) *case-prodD mem-Collect-eq*)
**then show** *?thesis* **by** *simp*
**qed**

**lemma** *darcs-neq-if-dtrees-neq*:
⟦*Node r xs* = *to-dtree-aux r*; (*x*,*e1*) ∈ *fset xs*; (*y*,*e2*) ∈ *fset xs*; *x*≠*y*⟧ $\Longrightarrow$ *e1* ≠ *e2*
**using** *dtree-children-fset-id* **by** (*metis* (*mono-tags*, *lifting*) *case-prodD mem-Collect-eq*)

**lemma** *dtrees-neq-if-darcs-neq*:
⟦*Node r xs* = *to-dtree-aux r*; (*x*,*e1*) ∈ *fset xs*; (*y*,*e2*) ∈ *fset xs*; *e1*≠*e2*⟧ $\Longrightarrow$ *x* ≠ *y*
**using** *dtree-children-fset-id case-prodD dtree-aux-fst-head-snd fst-conv*
**by** (*metis* (*no-types*, *lifting*) *mem-Collect-eq out-arcs-in-arcs snd-conv two-in-arcs-contr*)

**lemma** *dverts-disjoint*:
**assumes** *Node r xs* = *to-dtree-aux r* **and** (*x*,*e1*) ∈ *fset xs* **and** (*y*,*e2*) ∈ *fset xs*

```
    and (x,e1)≠(y,e2)
  shows dverts x ∩ dverts y = {}
proof (rule ccontr)
  assume dverts x ∩ dverts y ≠ {}
  then obtain v where v-def: v ∈ dverts x ∧ v ∈ dverts y by blast
  have x ≠ y using dtrees-neq-if-darcs-neq assms by blast
  have 0: x = to-dtree-aux (Dtree.root x) using to-dtree-aux-self assms(1,2) by
blast
  have 1: r →T Dtree.root x
   using assms(1,2) dominated-if-child by (metis (no-types, opaque-lifting) fst-conv
image-iff)
  then have 2: Dtree.root x ∈ verts T using adj-in-verts(2) by simp
  have 3: y = to-dtree-aux (Dtree.root y) using to-dtree-aux-self assms(1,3) by
blast
  have 4: r →T Dtree.root y
   using assms(1,3) dominated-if-child by (metis (no-types, opaque-lifting) fst-conv
image-iff)
  then have 5: Dtree.root y ∈ verts T using adj-in-verts(2) by simp
  have Dtree.root x →*T v using 0 2 to-dtree-aux-dverts-reachable v-def by blast
 moreover have Dtree.root y →*T v using 3 5 to-dtree-aux-dverts-reachable v-def
by blast
  moreover have Dtree.root x ≠ Dtree.root y using 0 3 assms(4) ‹x≠y› by auto
  ultimately show False using 1 4 reachable-via-child-impl-same by simp
qed

lemma wf-dverts-to-dtree-aux1: r ∉ verts T ⟹ wf-dverts (to-dtree-aux r)
  using to-dtree-aux-empty-if-notT unfolding wf-dverts-iff-dverts' by simp

lemma wf-dverts-to-dtree-aux2: r ∈ verts T ⟹ t = to-dtree-aux r ⟹ wf-dverts
t
proof(induction t arbitrary: r rule: darcs-mset.induct)
  case (1 r' xs)
  then have r = r' by simp
  have ∀(x,e) ∈ fset xs. wf-dverts x ∧ r ∉ dverts x
  proof (standard, standard, standard)
    fix xp x e
    assume asm: xp ∈ fset xs xp = (x,e)
   then have 0: x = to-dtree-aux (Dtree.root x) using to-dtree-aux-self 1.prems(2)
by simp
    have 2: r →T Dtree.root x using asm 1.prems ‹r = r'›
      by (metis (no-types, opaque-lifting) dominated-if-child fst-conv image-iff)
    then have 3: Dtree.root x ∈ verts T using adj-in-verts(2) by simp
    then show wf-dverts x using 1.IH asm 0 by blast
    show r ∉ dverts x
    proof
      assume r ∈ dverts x
       then have Dtree.root x →*T r using 0 3 to-dtree-aux-dverts-reachable by
blast
      then have r →+T r using 2 by auto
```

```
    then show False using reachable1-not-reverse by blast
   qed
  qed
 then show ?case using dverts-disjoint ‹r=r'› 1.prems(1,2) unfolding wf-dverts-iff-dverts'
   by (smt (verit, del-insts) wf-dverts'.simps case-prodI2 case-prod-conv)
qed

lemma wf-dverts-to-dtree-aux: wf-dverts (to-dtree-aux r)
 using wf-dverts-to-dtree-aux1 wf-dverts-to-dtree-aux2 by blast

lemma wf-dverts-to-dtree-aux': t = to-dtree-aux r ⟹ wf-dverts t
 using wf-dverts-to-dtree-aux by blast

lemma wf-dverts-to-dtree: wf-dverts to-dtree
 using to-dtree-def wf-dverts-to-dtree-aux by simp

lemma darcs-not-in-subtree:
 assumes Node r xs = to-dtree-aux r and (x,e) ∈ fset xs and (y,e2) ∈ fset xs
 shows e ∉ darcs y
proof
 assume asm: e ∈ darcs y
 have 0: y = to-dtree-aux (Dtree.root y) using to-dtree-aux-self assms(1,3) by
blast
 then obtain v where v-def: v ∈ dverts y ∧ e ∈ out-arcs T v using darcs-in-out-arcs
asm by blast
 have 1: r →_T Dtree.root y
  using assms(1,3) by (metis (no-types, opaque-lifting) dominated-if-child fst-conv
image-iff)
 then have Dtree.root y ∈ verts T using adj-in-verts(2) by auto
 then have Dtree.root y →*_T v using to-dtree-aux-dverts-reachable 0 v-def by
blast
 then have r →+_T v using 1 by auto
 then have r ≠ v using reachable1-not-reverse two-in-arcs-contr by blast
 moreover have tail T e = v using v-def by simp
 moreover have tail T e = r using assms(1,2) child-darc-tail-parent by blast
 ultimately show False by blast
qed

lemma darcs-disjoint:
 assumes Node r xs = to-dtree-aux r and r ∈ verts T
    and (x,e1) ∈ fset xs and (y,e2) ∈ fset xs and (x,e1)≠(y,e2)
 shows (darcs x ∪ {e1}) ∩ (darcs y ∪ {e2}) = {}
proof (rule ccontr)
 assume (darcs x ∪ {e1}) ∩ (darcs y ∪ {e2}) ≠ {}
 moreover have e1 ∉ darcs y using darcs-not-in-subtree assms(1−4) by blast
 moreover have e2 ∉ darcs x using darcs-not-in-subtree assms(1−4) by blast
 moreover have e1 ≠ e2 using darcs-neq-if-dtrees-neq assms by blast
 ultimately have darcs x ∩ darcs y ≠ {} by blast
 then obtain e where e-def: e ∈ darcs x ∧ e ∈ darcs y by blast
```

**have** $x$ = *to-dtree-aux* (*Dtree.root x*) **using** *to-dtree-aux-self assms*(*1*,*3*) **by** *blast*
**then obtain** *v1* **where** *v1-def*: *v1* ∈ *dverts x* ∧ *e* ∈ *out-arcs T v1*
**using** *darcs-in-out-arcs e-def* **by** *blast*
**have** $y$ = *to-dtree-aux* (*Dtree.root y*) **using** *to-dtree-aux-self assms*(*1*,*4*) **by** *blast*
**then obtain** *v2* **where** *v2-def*: *v2* ∈ *dverts y* ∧ *e* ∈ *out-arcs T v2*
**using** *darcs-in-out-arcs e-def* **by** *blast*
**then have** *v2* ≠ *v1* **using** *v1-def v2-def dverts-disjoint assms dtrees-neq-if-darcs-neq*
**by** *blast*
**then show** *False* **using** *v1-def v2-def* **by** *simp*
**qed**

**lemma** *wf-darcs-to-dtree-aux1*: *r* ∉ *verts T* ⟹ *wf-darcs* (*to-dtree-aux r*)
**using** *to-dtree-aux-empty-if-notT* **unfolding** *wf-darcs-def* **by** *simp*

**lemma** *wf-darcs-to-dtree-aux2*: *r* ∈ *verts T* ⟹ *t* = *to-dtree-aux r* ⟹ *wf-darcs t*
**proof**(*induction t arbitrary*: *r rule*: *darcs-mset.induct*)
**case** (*1 r′ xs*)
**then have** *r* = *r′* **by** *simp*
**have** ∀(*x*,*e*) ∈ *fset xs*. *wf-darcs x*
**proof** (*standard*, *standard*)
**fix** *xp x e*
**assume** *asm*: *xp* ∈ *fset xs xp* = (*x*,*e*)
**then have** *0*: *x* = *to-dtree-aux* (*Dtree.root x*) **using** *to-dtree-aux-self 1.prems*(*2*)
**by** *simp*
**have** *r* →$_T$ *Dtree.root x* **using** *asm 1.prems* ‹*r* = *r′*›
**by** (*metis* (*no-types*, *opaque-lifting*) *dominated-if-child fst-conv image-iff*)
**then have** *Dtree.root x* ∈ *verts T* **using** *adj-in-verts*(*2*) **by** *simp*
**then show** *wf-darcs x* **using** *1.IH asm 0* **by** *blast*
**qed**
**moreover have** ∀(*x*,*e1*) ∈ *fset xs*. (∀(*y*,*e2*) ∈ *fset xs*.
(*darcs x* ∪ {*e1*}) ∩ (*darcs y* ∪ {*e2*}) = {} ∨ (*x*,*e1*)=(*y*,*e2*))
**using** *darcs-disjoint 1.prems* ‹*r* = *r′*› **by** *blast*
**ultimately show** *?case* **using** *darcs-not-in-subtree 1.prems* ‹*r* = *r′*›
**by** (*smt* (*verit*) *case-prodD case-prodI2 wf-darcs-if-darcs′-aux*)
**qed**

**lemma** *wf-darcs-to-dtree-aux*: *wf-darcs* (*to-dtree-aux r*)
**using** *wf-darcs-to-dtree-aux1 wf-darcs-to-dtree-aux2* **by** *blast*

**lemma** *wf-darcs-to-dtree-aux′*: *t* = *to-dtree-aux r* ⟹ *wf-darcs t*
**using** *wf-darcs-to-dtree-aux* **by** *blast*

**lemma** *wf-darcs-to-dtree*: *wf-darcs to-dtree*
**using** *to-dtree-def wf-darcs-to-dtree-aux* **by** *simp*

**lemma** *dtail-aux-elem-eq-tail*:
*t* = *to-dtree-aux r* ⟹ *e* ∈ *darcs t* ⟹ *dtail t def e* = *tail T e*
**proof**(*induction t arbitrary*: *r rule*: *darcs-mset.induct*)
**case** (*1 r′ xs*)

```
  then have r = r' by simp
  let ?f = (λ(x,e2) b. if (x,e2) ∉ fset xs ∨ e ∉ darcs x ∨ ¬disjoint-darcs xs
            then b else dtail x def)
  show ?case
  proof(cases e ∈ snd ` fset xs)
    case True
    then have 0: dtail (Node r' xs) def e = r using ‹r=r'› by simp
    have e ∈ out-arcs T r using dtree-out-arcs-eq-snd 1.prems(1) True by simp
    then have tail T e = r by simp
    then show ?thesis using 0 by blast
  next
    case False
   then obtain x e1 where x-def: (x,e1) ∈ fset xs ∧ e ∈ darcs x using 1.prems(2)
by force
   then have x = to-dtree-aux (Dtree.root x) using 1.prems(1) ‹r = r'› to-dtree-aux-self
by blast
    then have 0: dtail x def e = tail T e using 1.IH x-def by blast
    have wf-darcs (Node r xs) using 1.prems(1) wf-darcs-to-dtree-aux by simp
    then have dtail (Node r' xs) def e = dtail x def e
      using dtail-in-child-eq-child[of x] x-def 1.prems by force
    then show ?thesis using 0 by simp
  qed
qed

lemma dtail-elem-eq-tail: e ∈ darcs to-dtree ⟹ dtail to-dtree def e = tail T e
  using dtail-aux-elem-eq-tail to-dtree-def by blast

lemma to-dtree-dtail-eq-tail-aux: dtail to-dtree (tail T) e = tail T e
  using dtail-notelem-eq-def dtail-elem-eq-tail by metis

lemma to-dtree-dtail-eq-tail: dtail to-dtree (tail T) = tail T
  using to-dtree-dtail-eq-tail-aux by blast

lemma dhead-aux-elem-eq-head:
  t = to-dtree-aux r ⟹ e ∈ darcs t ⟹ dhead t def e = head T e
proof(induction t arbitrary: r rule: darcs-mset.induct)
  case (1 r' xs)
  then have r = r' by simp
  let ?f = (λ(x,e2) b. if (x,e2) ∉ fset xs ∨ e ∉ (darcs x ∪ {e2}) ∨ ¬disjoint-darcs
xs
           then b else if e=e2 then Dtree.root x else dhead x def e)
  obtain child where child ∈ fset xs using 1.prems(2) by auto
  then have wf: wf-darcs (Node r xs) using 1.prems(1) wf-darcs-to-dtree-aux by
simp
  show ?case
  proof(cases e ∈ snd ` fset xs)
    case True
    then obtain x where x-def: (x,e) ∈ fset xs by force
    then have 0: dhead (Node r' xs) def e = Dtree.root x
```

**using** *dhead-in-set-eq-root wf* ‹*r*=*r'*› **by** *fast*
**have** *e* ∈ *out-arcs T r* **using** *dtree-out-arcs-eq-snd 1.prems*(*1*) *True* **by** *simp*
**then have** *head T e* = *Dtree.root x* **using** *x-def 1.prems*(*1*) *dtree-aux-fst-head-snd*
**by** *force*
**then show** *?thesis* **using** *0* **by** *simp*
**next**
**case** *False*
**then obtain** *x e1* **where** *x-def*: (*x*,*e1*) ∈ *fset xs* ∧ *e* ∈ *darcs x* **using** *1.prems*(*2*)
**by** *force*
**then have** *x* = *to-dtree-aux* (*Dtree.root x*) **using** *1.prems*(*1*) ‹*r* = *r'*› *to-dtree-aux-self*
**by** *blast*
**then have** *0*: *dhead x def e* = *head T e* **using** *1.IH x-def* **by** *blast*
**have** *dhead* (*Node r' xs*) *def e* = *dhead x def e*
**using** *dhead-in-child-eq-child*[*of x*] *x-def wf* ‹*r*=*r'*› **by** *blast*
**then show** *?thesis* **using** *0* **by** *simp*
**qed**
**qed**

**lemma** *dhead-elem-eq-head*: *e* ∈ *darcs to-dtree* ⟹ *dhead to-dtree def e* = *head T*
*e*
**using** *dhead-aux-elem-eq-head to-dtree-def* **by** *blast*

**lemma** *to-dtree-dhead-eq-head-aux*: *dhead to-dtree* (*head T*) *e* = *head T e*
**using** *dhead-notelem-eq-def dhead-elem-eq-head* **by** *metis*

**lemma** *to-dtree-dhead-eq-head*: *dhead to-dtree* (*head T*) = *head T*
**using** *to-dtree-dhead-eq-head-aux* **by** *blast*

**lemma** *from-to-dtree-eq-orig*: *from-dtree* (*to-dtree*) = *T*
**using** *to-dtree-dhead-eq-head to-dtree-dtail-eq-tail darcs-eq-arcs dverts-eq-verts* **by**
*simp*

**lemma** *subtree-darc-tail-parent*:
⟦*is-subtree* (*Node r xs*) *to-dtree*; (*t*,*e*) ∈ *fset xs*⟧ ⟹ *tail T e* = *r*
**using** *child-darc-tail-parent to-dtree-self-subtree'* **by** *blast*

**lemma** *subtree-darc-head-root*:
⟦*is-subtree* (*Node r xs*) *to-dtree*; (*t*,*e*) ∈ *fset xs*⟧ ⟹ *head T e* = *Dtree.root t*
**using** *child-darc-head-root to-dtree-self-subtree'* **by** *blast*

**lemma** *subtree-darc-in-arcs*:
⟦*is-subtree* (*Node r xs*) *to-dtree*; (*t*,*e*) ∈ *fset xs*⟧ ⟹ *e* ∈ *arcs T*
**using** *to-dtree-self-subtree' child-darc-in-arcs* **by** *blast*

**lemma** *subtree-child-dom*: ⟦*is-subtree* (*Node r xs*) *to-dtree*; (*t*,*e*) ∈ *fset xs*⟧ ⟹ *r*
→$_T$ *Dtree.root t*
**using** *subtree-darc-tail-parent subtree-darc-head-root subtree-darc-in-arcs*
*in-arcs-imp-in-arcs-ends* **by** *fastforce*

**end**

### 7.3.2 Well-Formed Dtrees

**locale** *wf-dtree* =
  **fixes** *t* :: ('*a*,'*b*) *dtree*
  **assumes** *wf-arcs*: *wf-darcs t*
    **and** *wf-verts*: *wf-dverts t*

**begin**

**lemma** *wf-dtree-rec*: *Node r xs* = *t* ⟹ (*x*,*e*) ∈ *fset xs* ⟹ *wf-dtree x*
  **using** *wf-arcs wf-verts* **by** (*unfold-locales*) *auto*

**lemma** *wf-dtree-sub*: *is-subtree x t* ⟹ *wf-dtree x*
**using** *wf-dtree-axioms* **proof**(*induction t rule*: *darcs-mset.induct*)
  **case** (*1 r xs*)
  **then interpret** *wf-dtree Node r xs* **by** *blast*
  **show** *?case*
  **proof**(*cases x* = *Node r xs*)
    **case** *True*
    **then show** *?thesis* **by** (*simp add*: *wf-dtree-axioms*)
  **next**
    **case** *False*
    **then show** *?thesis* **using** *1.IH wf-dtree-rec 1.prems*(*1*) **by** *auto*
  **qed**
**qed**

**lemma** *root-not-subtree*: ⟦(*Node r xs*) = *t*; *x* ∈ *fst* ‘ *fset xs*⟧ ⟹ *r* ∉ *dverts x*
  **using** *wf-verts root-not-child-if-wf-dverts* **by** *fastforce*

**lemma** *dverts-child-subset*: ⟦(*Node r xs*) = *t*; *x* ∈ *fst* ‘ *fset xs*⟧ ⟹ *dverts x* ⊂ *dverts t*
  **using** *root-not-subtree* **by** *fastforce*

**lemma** *child-arc-not-subtree*: ⟦(*Node r xs*) = *t*; (*x*,*e1*) ∈ *fset xs*⟧ ⟹ *e1* ∉ *darcs x*
  **using** *wf-arcs disjoint-darcs-if-wf-aux3* **by** *fast*

**lemma** *darcs-child-subset*: ⟦(*Node r xs*) = *t*; *x* ∈ *fst* ‘ *fset xs*⟧ ⟹ *darcs x* ⊂ *darcs t*
  **using** *child-arc-not-subtree* **by** *force*

**lemma** *dtail-in-dverts*: *e* ∈ *darcs t* ⟹ *dtail t def e* ∈ *dverts t*
**using** *wf-arcs* **proof**(*induction t rule*: *darcs-mset.induct*)
  **case** (*1 r xs*)
  **show** *?case*
  **proof**(*cases e* ∈ *snd* ‘ *fset xs*)
    **case** *False*
   **then obtain** *x e1* **where** *x-def*: (*x*,*e1*) ∈ *fset xs* ∧ *e* ∈ *darcs x* **using** *1.prems*(*1*)

**by** *force*
  **then have** *wf-darcs x* **using** *1.prems(2)* **by** *auto*
  **then have** *dtail x def e* ∈ *dverts x* **using** *1.IH x-def* **by** *blast*
  **then have** *0*: *dtail x def e* ∈ *dverts* (*Node r xs*)
    **using** *x-def* **by** (*auto simp*: *dverts-child-subseteq*)
  **have** *dtail* (*Node r xs*) *def e* = *dtail x def e*
    **using** *dtail-in-child-eq-child*[*of x*] *x-def 1.prems(2)* **by** *blast*
  **then show** *?thesis* **using** *0* **by** *argo*
 **qed** (*simp*)
**qed**

**lemma** *dtail-in-childverts*:
 **assumes** *e* ∈ *darcs x* **and** (*x*,*e'*) ∈ *fset xs* **and** *Node r xs* = *t*
 **shows** *dtail t def e* ∈ *dverts x*
**proof** −
 **interpret** *X*: *wf-dtree x* **using** *assms*(*2*,*3*) *wf-dtree-rec* **by** *blast*
 **have** *dtail t def e* = *dtail x def e*
  **using** *dtail-in-child-eq-child*[*of x*] *assms wf-arcs* **by** *force*
 **then show** *?thesis* **using** *assms*(*1*) *X.dtail-in-dverts* **by** *simp*
**qed**

**lemma** *dhead-in-dverts*: *e* ∈ *darcs t* ⟹ *dhead t def e* ∈ *dverts t*
**using** *wf-arcs* **proof**(*induction t rule*: *darcs-mset.induct*)
 **case** (*1 r xs*)
 **show** *?case*
 **proof**(*cases e* ∈ *snd* ‘ *fset xs*)
  **case** *True*
  **then obtain** *x* **where** *x-def*: (*x*,*e*) ∈ *fset xs* **by** *force*
  **then have** *dhead* (*Node r xs*) *def e* = *root x*
    **using** *dhead-in-set-eq-root*[*of x*] *1.prems(2)* **by** *blast*
  **then show** *?thesis* **using** *dtree.set-sel*(*1*) *x-def* **by** *fastforce*
 **next**
  **case** *False*
  **then obtain** *x e1* **where** *x-def*: (*x*,*e1*) ∈ *fset xs* ∧ *e* ∈ *darcs x* **using** *1.prems*(*1*)
**by** *force*
  **then have** *wf-darcs x* **using** *1.prems(2)* **by** *auto*
  **then have** *dhead x def e* ∈ *dverts x* **using** *1.IH x-def* **by** *blast*
  **then have** *0*: *dhead x def e* ∈ *dverts* (*Node r xs*)
    **using** *x-def* **by** (*auto simp*: *dverts-child-subseteq*)
  **have** *dhead* (*Node r xs*) *def e* = *dhead x def e*
    **using** *dhead-in-child-eq-child*[*of x*] *x-def 1.prems(2)* **by** *force*
  **then show** *?thesis* **using** *0* **by** *argo*
 **qed**
**qed**

**lemma** *dhead-in-childverts*:
 **assumes** *e* ∈ *darcs x* **and** (*x*,*e'*) ∈ *fset xs* **and** *Node r xs* = *t*
 **shows** *dhead t def e* ∈ *dverts x*
**proof** −

**interpret** $X$: *wf-dtree x* **using** *wf-arcs wf-verts assms*(*2*,*3*) **by**(*unfold-locales*) *auto*
**have** *dhead t def e* = *dhead x def e*
**using** *dhead-in-child-eq-child*[*of x*] *assms wf-arcs* **by** *auto*
**then show** *?thesis* **using** *assms*(*1*) *X.dhead-in-dverts* **by** *simp*
**qed**

**lemma** *dhead-in-dverts-no-root*: $e \in$ *darcs t* $\Longrightarrow$ *dhead t def e* $\in$ (*dverts t* $-$ {*root t*})
**using** *wf-arcs wf-verts* **proof**(*induction t rule*: *darcs-mset.induct*)
**case** (*1 r xs*)
**interpret** *wf-dtree Node r xs* **using** *1.prems*(*2*,*3*) **by** (*unfold-locales*) *auto*
**show** *?case*
**proof**(*cases e* $\in$ *snd* ‘ *fset xs*)
**case** *True*
**then obtain** *x* **where** *x-def*: (*x*,*e*) $\in$ *fset xs* **by** *force*
**then have** *dhead* (*Node r xs*) *def e* = *root x*
**using** *dhead-in-set-eq-root*[*of x*] *1.prems*(*2*) **by** *simp*
**then show** *?thesis* **using** *dtree.set-sel*(*1*) *x-def 1.prems*(*3*) *wf-dverts-iff-dverts′* **by** *fastforce*
**next**
**case** *False*
**then obtain** *x e1* **where** *x-def*: (*x*,*e1*) $\in$ *fset xs* $\wedge$ *e* $\in$ *darcs x* **using** *1.prems*(*1*) **by** *force*
**then have** *wf-darcs x* **using** *1.prems*(*2*) **by** *auto*
**then have** *dhead x def e* $\in$ *dverts x* **using** *1.IH x-def 1.prems*(*3*) **by** *auto*
**moreover have** *r* $\notin$ *dverts x* **using** *root-not-subtree x-def* **by** *fastforce*
**ultimately have** *0*: *dhead x def e* $\in$ *dverts* (*Node r xs*) $-$ {*root* (*Node r xs*)}
**using** *x-def dverts-child-subseteq* **by** *fastforce*
**have** *dhead* (*Node r xs*) *def e* = *dhead x def e*
**using** *dhead-in-child-eq-child*[*of x*] *x-def 1.prems*(*2*) **by** *force*
**then show** *?thesis* **using** *0* **by** *argo*
**qed**
**qed**

**lemma** *dhead-in-childverts-no-root*:
**assumes** *e* $\in$ *darcs x* **and** (*x*,*e′*) $\in$ *fset xs* **and** *Node r xs* = *t*
**shows** *dhead t def e* $\in$ (*dverts x* $-$ {*root x*})
**proof** $-$
**interpret** $X$: *wf-dtree x* **using** *assms*(*2*,*3*) *wf-dtree-rec* **by** *blast*
**have** *dhead t def e* = *dhead x def e*
**using** *dhead-in-child-eq-child*[*of x*] *assms wf-arcs* **by** *auto*
**then show** *?thesis* **using** *assms*(*1*) *X.dhead-in-dverts-no-root* **by** *simp*
**qed**

**lemma** *dtree-cas-iff-subtree*:
**assumes** (*x*,*e1*) $\in$ *fset xs* **and** *Node r xs* = *t* **and** *set p* $\subseteq$ *darcs x*
**shows** *pre-digraph.cas* (*from-dtree dt dh x*) *u p v*
$\longleftrightarrow$ *pre-digraph.cas* (*from-dtree dt dh t*) *u p v*

```
    (is pre-digraph.cas ?X - - - ⟷ pre-digraph.cas ?T - - -)
using assms proof(induction p arbitrary: u)
  case Nil
  then show ?case by(simp add: pre-digraph.cas.simps(1))
next
  case (Cons p ps)
  note pre-digraph.cas.simps[simp]
  have pre-digraph.cas ?T u (p # ps) v = (tail ?T p = u ∧ pre-digraph.cas ?T
(head ?T p) ps v)
    by simp
  also have ... = (tail ?T p = u ∧ pre-digraph.cas ?X (head ?T p) ps v)
    using Cons.IH Cons.prems by simp
  also have ... = (tail ?X p = u ∧ pre-digraph.cas ?X (head ?T p) ps v)
    using dtail-in-child-eq-child[of x] Cons.prems(1−3) wf-arcs by force
  also have ... = (tail ?X p = u ∧ pre-digraph.cas ?X (head ?X p) ps v)
    using dhead-in-child-eq-child[of x] Cons.prems(1−3) wf-arcs by force
  finally show ?case by simp
qed

lemma dtree-cas-exists:
  v ∈ dverts t ⟹ ∃ p. set p ⊆ darcs t ∧ pre-digraph.cas (from-dtree dt dh t) (root
t) p v
using wf-dtree-axioms proof(induction t)
  case (Node r xs)
  then show ?case
  proof(cases r=v)
    case True
    then have pre-digraph.cas (from-dtree dt dh (Node r xs)) (root (Node r xs)) []
v
      by (simp add: pre-digraph.cas.simps(1))
    then show ?thesis by force
  next
    case False
   then obtain x e where x-def: (x,e) ∈ fset xs ∧ v ∈ dverts x using Node.prems
by auto
    let ?T = from-dtree dt dh (Node r xs)
    let ?X = from-dtree dt dh x
    interpret wf-dtree Node r xs by (rule Node.prems(2))
    have wf-dtree x using x-def wf-dtree-rec by blast
   then obtain p where p-def: set p ⊆ darcs x ∧ pre-digraph.cas ?X (root x) p v
      using Node.IH x-def by fastforce
    then have pre-digraph.cas ?T (root x) p v
      using dtree-cas-iff-subtree x-def Node.prems(2) by blast
    moreover have head ?T e = root x
      using x-def dhead-in-set-eq-root[of x] wf-arcs by simp
    moreover have tail ?T e = r using x-def by force
    ultimately have pre-digraph.cas ?T (root (Node r xs)) (e#p) v
      by (simp add: pre-digraph.cas.simps(2))
    moreover have set (e#p) ⊆ darcs (Node r xs) using p-def x-def by force
```

**ultimately show** *?thesis* **by** *blast*
**qed**
**qed**

**lemma** *dtree-awalk-exists*:
**assumes** $v \in$ *dverts t*
**shows** $\exists p$. *pre-digraph.awalk* (*from-dtree dt dh t*) (*root t*) *p v*
**unfolding** *pre-digraph.awalk-def* **using** *dtree-cas-exists assms dtree.set-sel*(*1*) **by** *fastforce*

**lemma** *subtree-root-not-root*: $t =$ *Node r xs* $\Longrightarrow (x,e) \in$ *fset xs* $\Longrightarrow$ *root x* $\neq r$
**using** *dtree.set-sel*(*1*) *root-not-subtree* **by** *fastforce*

**lemma** *dhead-not-root*:
**assumes** $e \in$ *darcs t*
**shows** *dhead t def e* $\neq$ *root t*
**proof** −
**obtain** *r xs* **where** *xs-def*[*simp*]: $t =$ *Node r xs* **using** *dtree.exhaust* **by** *auto*
**show** *?thesis*
**proof**(*cases* $e \in$ *snd* ‘ *fset xs*)
**case** *True*
**then obtain** *x* **where** *x-def*: $(x,e) \in$ *fset xs* **by** *force*
**then have** *dhead* (*Node r xs*) *def e* = *root x*
**using** *dhead-in-set-eq-root*[*of x*] *wf-arcs* **by** *simp*
**then show** *?thesis* **using** *x-def subtree-root-not-root* **by** *simp*
**next**
**case** *False*
**then obtain** *x e1* **where** *x-def*: $(x,e1) \in$ *fset xs* $\wedge$ $e \in$ *darcs x* **using** *assms*
**by** *force*
**then interpret** *X*: *wf-dtree x* **using** *wf-dtree-rec* **by** *auto*
**have** *dhead x def e* $\in$ *dverts x* **using** *x-def X.dhead-in-dverts* **by** *blast*
**moreover have** *dhead* (*Node r xs*) *def e* = *dhead x def e*
**using** *x-def dhead-in-child-eq-child*[*of x*] *wf-arcs* **by** *force*
**ultimately show** *?thesis* **using** *x-def root-not-subtree* **by** *fastforce*
**qed**
**qed**

**lemma** *nohead-cas-no-arc-in-subset*:
⟦$\forall e \in$ *darcs t*. *dhead t dh e* $\neq v$; *p*$\neq$[]; *pre-digraph.cas* (*from-dtree dt dh t*) *u p v*⟧
$\Longrightarrow \neg$*set p* $\subseteq$ *darcs t*
**by**(*induction p arbitrary*: *u*) (*fastforce simp*: *pre-digraph.cas.simps*)+

**lemma** *dtail-root-in-set*:
**assumes** $e \in$ *darcs t* **and** $t =$ *Node r xs* **and** *dtail t dt e* = *r*
**shows** $e \in$ *snd* ‘ *fset xs*
**proof** (*rule ccontr*)
**assume** $e \notin$ *snd* ‘ *fset xs*
**then obtain** *x e1* **where** *x-def*: $(x,e1) \in$ *fset xs* $\wedge$ $e \in$ *darcs x* **using** *assms*(*1*,*2*)
**by** *force*

**interpret** *X*: *wf-dtree x* **using** *assms*(*2*) *x-def wf-dtree-rec* **by** *blast*
**have** *dtail t dt e* = *dtail x dt e*
**using** *dtail-in-child-eq-child*[*of x*] *wf-arcs assms*(*2*) *x-def* **by** *force*
**then have** *dtail t dt e* ∈ *dverts x* **using** *X.dtail-in-dverts x-def* **by** *simp*
**then show** *False* **using** *assms*(*2*,*3*) *wf-verts x-def* **unfolding** *wf-dverts-iff-dverts′*
**by** *auto*
**qed**

**lemma** *dhead-notin-subtree-wo-root*:
**assumes** (*x*,*e*) ∈ *fset xs* **and** *p* ∉ *darcs x* **and** *p* ∈ *darcs t* **and** *t* = *Node r xs*
**shows** *dhead t dh p* ∉ (*dverts x* − {*root x*})
**proof**(*cases p* ∈ *snd* ‘ *fset xs*)
**case** *True*
**then obtain** *x′* **where** *x′-def*: (*x′*,*p*) ∈ *fset xs* **by** *auto*
**then have** *0*: *dhead t dh p* = *root x′*
**using** *dhead-in-set-eq-root*[*of x′*] *wf-arcs assms*(*4*) **by** *auto*
**have** *root x′* ∉ (*dverts x* − {*root x*})
**proof**(*cases x′*=*x*)
**case** *True*
**then show** *?thesis* **by** *blast*
**next**
**case** *False*
**have** *root x′* ∈ *dverts x′* **by** (*simp add*: *dtree.set-sel*(*1*))
**then show** *?thesis* **using** *wf-verts x′-def assms*(*1*,*4*) **unfolding** *wf-dverts-iff-dverts′*
**by** *fastforce*
**qed**
**then show** *?thesis* **using** *0* **by** *simp*
**next**
**case** *False*
**then obtain** *x′ e1* **where** *x′-def*: (*x′*,*e1*) ∈ *fset xs* ∧ *p* ∈ *darcs x′* **using**
*assms*(*3*,*4*) **by** *force*
**then have** *0*: *dhead t dh p* = *dhead x′ dh p*
**using** *dhead-in-child-eq-child*[*of x′*] *wf-arcs assms*(*4*) **by** *auto*
**interpret** *X*: *wf-dtree x′* **using** *assms*(*4*) *x′-def wf-dtree-rec* **by** *blast*
**have** *1*: *dhead x′ dh p* ∈ *dverts x′* **using** *X.dhead-in-dverts x′-def* **by** *blast*
**moreover have** *dverts x′* ∩ *dverts x* = {}
**using** *wf-verts x′-def assms*(*1*,*2*,*4*) **unfolding** *wf-dverts-iff-dverts′* **by** *fastforce*
**ultimately show** *?thesis* **using** *0* **by** *auto*
**qed**

**lemma** *subtree-uneq-if-arc-uneq*:
⟦(*x1*,*e1*) ∈ *fset xs*; (*x2*,*e2*) ∈ *fset xs*; *e1*≠*e2*; *Node r xs* = *t*⟧ ⟹ *x1* ≠ *x2*
**using** *dtree.set-sel*(*1*) *wf-verts disjoint-dverts-if-wf-aux* **by** *fast*

**lemma** *arc-uneq-if-subtree-uneq*:
⟦(*x1*,*e1*) ∈ *fset xs*; (*x2*,*e2*) ∈ *fset xs*; *x1*≠*x2*; *Node r xs* = *t*⟧ ⟹ *e1* ≠ *e2*
**using** *disjoint-darcs-if-wf*[*OF wf-arcs*] **by** *fastforce*

**lemma** *dhead-unique*: *e* ∈ *darcs t* ⟹ *p* ∈ *darcs t* ⟹ *e* ≠ *p* ⟹ *dhead t dh e* ≠

*dhead t dh p*
**using** *wf-dtree-axioms* **proof**(*induction t rule*: *darcs-mset.induct*)
**case** *ind*: (*1 r xs*)
**then interpret** *wf-dtree Node r xs* **by** *blast*
**show** *?case*
**proof**(*cases* ∃ *x* ∈ *fst* ‘ *fset xs*. *e* ∈ *darcs x* ∧ *p* ∈ *darcs x*)
**case** *True*
**then obtain** *x e1* **where** *x-def*: (*x*,*e1*) ∈ *fset xs* ∧ *e* ∈ *darcs x* ∧ *p* ∈ *darcs x*
**by** *force*
**then have** *wf-dtree x* **using** *ind.prems*(*4*) *wf-dtree-rec* **by** *blast*
**then have** *dhead x dh e* ≠ *dhead x dh p* **using** *ind x-def* **by** *blast*
**then show** *?thesis* **using** *True dhead-in-child-eq-child*[*of x*] *wf-arcs x-def* **by**
*force*
**next**
**case** *False*
**then consider** ∃ *x* ∈ *fst* ‘ *fset xs*. *e* ∈ *darcs x* | ∃ *x* ∈ *fst* ‘ *fset xs*. *p* ∈ *darcs x*
| *e* ∈ *snd* ‘ *fset xs* ∧ *p* ∈ *snd* ‘ *fset xs*
**using** *ind.prems*(*1*,*2*) **by** *force*
**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then obtain** *x e1* **where** *x-def*: (*x*,*e1*) ∈ *fset xs* ∧ *e* ∈ *darcs x* ∧ *p* ∉ *darcs x*
**using** *False* **by** *force*
**then interpret** *X*: *wf-dtree x* **using** *wf-dtree-rec* **by** *blast*
**have** *dhead x dh e* ∈ (*dverts x* − {*root x*}) **using** *X.dhead-in-dverts-no-root*
*x-def* **by** *blast*
**then have** *dhead* (*Node r xs*) *dh e* ∈ (*dverts x* − {*root x*})
**using** *dhead-in-child-eq-child*[*of x*] *wf-arcs x-def* **by** *force*
**moreover have** *dhead* (*Node r xs*) *dh p* ∉ (*dverts x* − {*root x*})
**using** *x-def dhead-notin-subtree-wo-root ind.prems*(*2*) **by** *blast*
**ultimately show** *?thesis* **by** *auto*
**next**
**case** *2*
**then obtain** *x e1* **where** *x-def*: (*x*,*e1*) ∈ *fset xs* ∧ *p* ∈ *darcs x* ∧ *e* ∉ *darcs x*
**using** *False* **by** *force*
**then interpret** *X*: *wf-dtree x* **using** *wf-dtree-rec* **by** *blast*
**have** *dhead x dh p* ∈ (*dverts x* − {*root x*}) **using** *X.dhead-in-dverts-no-root*
*x-def* **by** *blast*
**then have** *dhead* (*Node r xs*) *dh p* ∈ (*dverts x* − {*root x*})
**using** *dhead-in-child-eq-child*[*of x*] *wf-arcs x-def* **by** *force*
**moreover have** *dhead* (*Node r xs*) *dh e* ∉ (*dverts x* − {*root x*})
**using** *x-def dhead-notin-subtree-wo-root ind.prems*(*1*) **by** *blast*
**ultimately show** *?thesis* **by** *auto*
**next**
**case** *3*
**then obtain** *x1 x2* **where** *x-def*: (*x1*,*p*) ∈ *fset xs* ∧ (*x2*,*e*) ∈ *fset xs* **by** *force*
**then have** *0*: *dhead* (*Node r xs*) *dh p* = *root x1* ∧ *dhead* (*Node r xs*) *dh e* =
*root x2*
**using** *dhead-in-set-eq-root*[*of x1*] *dhead-in-set-eq-root*[*of x2*] *wf-arcs* **by** *simp*

```
    have x1 ≠ x2 using subtree-uneq-if-arc-uneq x-def ind.prems(3) by blast
    then have root x1 ≠ root x2
        using wf-verts x-def dtree.set-sel(1) unfolding wf-dverts-iff-dverts' by
fastforce
    then show ?thesis using 0 by argo
   qed
  qed
qed

lemma arc-in-subtree-if-tail-in-subtree:
  assumes dtail t dt p ∈ dverts x
     and p ∈ darcs t
     and t = Node r xs
     and (x,e) ∈ fset xs
    shows p ∈ darcs x
proof (rule ccontr)
  assume asm: p ∉ darcs x
  show False
  proof(cases p ∈ snd ` fset xs)
    case True
    then have dtail t dt p = r using assms(2,3) by simp
    then show ?thesis using assms(1,3,4) root-not-subtree by force
  next
    case False
     then obtain x' e1 where x'-def: (x',e1) ∈ fset xs ∧ p ∈ darcs x' using
assms(2,3) by force
    then have x ≠ x' using asm by blast
    interpret X: wf-dtree x' using x'-def assms(3) wf-dtree-rec by blast
    have dtail t dt p = dtail x' dt p
      using dtail-in-child-eq-child[of x'] x'-def wf-arcs assms(3) by force
   then have dtail t dt p ∈ dverts x' using X.dtail-in-dverts by (simp add: x'-def)
    then have dtail t dt p ∉ dverts x
      using ‹x≠x'› wf-verts assms(3,4) x'-def unfolding wf-dverts-iff-dverts' by
fastforce
    then show ?thesis using assms(1) by blast
  qed
qed

lemma dhead-in-verts-if-dtail:
  assumes dtail t dt p ∈ dverts x
     and p ∈ darcs t
     and t = Node r xs
     and (x,e) ∈ fset xs
    shows dhead t dh p ∈ dverts x
proof −
  interpret X: wf-dtree x using assms(3,4) wf-dtree-rec by blast
  have 0: p ∈ darcs x using assms arc-in-subtree-if-tail-in-subtree by blast
  then have dhead t dh p = dhead x dh p
    using dhead-in-child-eq-child[of x] assms(3,4) wf-arcs by simp
```

**then show** *?thesis* **using** *X.dhead-in-dverts 0* **by** *simp*
**qed**

**lemma** *cas-darcs-in-subtree*:
**assumes** *pre-digraph.cas* (*from-dtree dt dh t*) *u ps v*
**and** *set ps* ⊆ *darcs t*
**and** *t* = *Node r xs*
**and** (*x*,*e*) ∈ *fset xs*
**and** *u* ∈ *dverts x*
**shows** *set ps* ⊆ *darcs x*
**using** *assms* **proof**(*induction ps arbitrary*: *u*)
**case** *Nil*
**then show** *?case* **by** *simp*
**next**
**case** (*Cons p ps*)
**note** *pre-digraph.cas.simps*[*simp*]
**then have** *u-p*: *dtail t dt p* = *u* **using** *Cons.prems*(*1*) **by** *simp*
**have** *p* ∈ *darcs t* **using** *Cons.prems*(*2*) **by** *simp*
**then have** *0*: *p* ∈ *darcs x* **using** *arc-in-subtree-if-tail-in-subtree Cons.prems*(*3*−*5*) *u-p* **by** *blast*
**have** *1*: *dhead t dh p* ∈ *dverts x* **using** *dhead-in-verts-if-dtail Cons.prems*(*2*−*5*) *u-p* **by** *force*
**have** *set ps* ⊆ *darcs t* **using** *Cons.prems*(*2*) **by** *simp*
**have** *pre-digraph.cas* (*from-dtree dt dh t*) (*dhead t dh p*) *ps v* **using** *Cons.prems*(*1*) **by** *simp*
**then have** *set ps* ⊆ *darcs x* **using** *Cons.IH Cons.prems*(*2*,*3*,*4*) *1* **by** *simp*
**then show** *?case* **using** *0* **by** *simp*
**qed**

**lemma** *dtree-cas-in-subtree*:
**assumes** *pre-digraph.cas* (*from-dtree dt dh t*) *u ps v*
**and** *set ps* ⊆ *darcs t*
**and** *t* = *Node r xs*
**and** (*x*,*e*) ∈ *fset xs*
**and** *u* ∈ *dverts x*
**shows** *pre-digraph.cas* (*from-dtree dt dh x*) *u ps v*
**using** *assms cas-darcs-in-subtree dtree-cas-iff-subtree* **by** *fast*

**lemma** *cas-to-end-subtree*:
**assumes** *set* (*p*#*ps*) ⊆ *darcs t* **and** *pre-digraph.cas* (*from-dtree dt dh t*) (*root t*) (*p*#*ps*) *v*
**and** *t* = *Node r xs* **and** (*x*,*e*) ∈ *fset xs* **and** *v* ∈ *dverts x*
**shows** *p* = *e*
**proof** (*rule ccontr*)
**assume** *asm*: *p* ≠ *e*
**note** *pre-digraph.cas.simps*[*simp*]
**have** *dtail t dt p* = *r* **using** *assms*(*2*,*3*) **by** *simp*
**then have** *p* ∈ *snd* ‘ *fset xs* **using** *dtail-root-in-set assms*(*1*,*3*) *list.set-intros*(*1*) **by** *fast*

```
  then obtain x' where x'-def: (x',p) ∈ fset xs by force
  show False
  proof(cases ps=[])
    case True
    then have root x' = v
      using dhead-in-set-eq-root[of x'] x'-def assms(2,3) wf-arcs by simp
    then have x = x'
    using wf-verts x'-def assms(3,4,5) dtree.set-sel(1) by (fastforce simp: wf-dverts-iff-dverts')
     then show ?thesis using asm assms(3,4) subtree-uneq-if-arc-uneq x'-def by
blast
  next
    case False
    interpret X: wf-dtree x' using wf-dtree-rec x'-def assms(3) by blast
    have x' ≠ x using asm assms(3,4) subtree-uneq-if-arc-uneq x'-def by blast
    then have x'-no-v: ∀ e∈darcs x'. dhead x' dh e ≠ v
      using X.dhead-in-dverts assms(3,4,5) x'-def wf-verts
      by (fastforce simp: wf-dverts-iff-dverts')
   have 0: pre-digraph.cas (from-dtree dt dh t) (dhead t dh p) ps v using assms(2)
by simp
    have 1: dhead t dh p ∈ dverts x'
      using dhead-in-set-eq-root[of x'] x'-def assms(3) dtree.set-sel(1) wf-arcs by
auto
    then have pre-digraph.cas (from-dtree dt dh x') (dhead t dh p) ps v
      using dtree-cas-in-subtree x'-def assms(1,3) 0 by force
     then have ¬ set ps ⊆ darcs x' using X.nohead-cas-no-arc-in-subset x'-no-v
False by blast
   moreover have set ps ⊆ darcs x' using cas-darcs-in-subtree assms(1,3) x'-def
0 1 by simp
    ultimately show ?thesis by blast
  qed
qed

lemma cas-unique-in-darcs: ⟦v ∈ dverts t; pre-digraph.cas (from-dtree dt dh t) (root
t) ps v;
       pre-digraph.cas (from-dtree dt dh t) (root t) es v⟧
      ⟹ ps = es ∨ ¬set ps ⊆ darcs t ∨ ¬set es ⊆ darcs t
using wf-dtree-axioms proof(induction t arbitrary: ps es rule: darcs-mset.induct)
  case ind: (1 r xs)
  interpret wf-dtree Node r xs by (rule ind.prems(4))
  show ?case
  proof(cases r=v)
    case True
   have 0: ∀ e ∈ darcs (Node r xs). dhead (Node r xs) dh e ≠ r using dhead-not-root
by force
    consider ps = [] ∧ es = [] | ps ≠ [] | es ≠ [] by blast
    then show ?thesis
    proof(cases)
      case 1
      then show ?thesis by blast
```

**next**
**case** *2*
**then show** *?thesis* **using** *nohead-cas-no-arc-in-subset 0 ind.prems(2) True* **by** *blast*
**next**
**case** *3*
**then show** *?thesis* **using** *nohead-cas-no-arc-in-subset 0 ind.prems(3) True* **by** *blast*
**qed**
**next**
**case** *False*
**then obtain** *x e* **where** *x-def*: $(x,e) \in$ *fset xs* $v \in$ *dverts x* **using** *ind.prems* **by** *auto*
**then have** *wf-x*: *wf-dtree x* **using** *wf-dtree-rec* **by** *blast*
**note** *pre-digraph.cas.simps*[*simp*]
**have** *nempty*: $ps \neq [] \wedge es \neq []$ **using** *ind.prems(2,3) False* **by** *force*
**then obtain** *p ps'* **where** *p-def*: $ps = p \# ps'$ **using** *list.exhaust-sel* **by** *auto*
**obtain** *e' es'* **where** *e'-def*: $es = e' \# es'$ **using** *list.exhaust-sel nempty* **by** *auto*
**show** *?thesis*
**proof** (*rule ccontr*)
**assume** $\neg(ps = es \vee \neg set\ ps \subseteq darcs\ (Node\ r\ xs) \vee \neg set\ es \subseteq darcs\ (Node\ r\ xs))$
**then have** *asm*: $ps \neq es \wedge set\ ps \subseteq darcs\ (Node\ r\ xs) \wedge set\ es \subseteq darcs\ (Node\ r\ xs)$ **by** *blast*
**then have** $p = e$ **using** *cas-to-end-subtree p-def ind.prems(2) x-def* **by** *blast*
**moreover have** $e' = e$ **using** *cas-to-end-subtree e'-def ind.prems(3) x-def asm* **by** *blast*
**ultimately have** $p = e'$ **by** *blast*
**have** *dhead* (*Node r xs*) *dh p* = *root x*
**using** *dhead-in-set-eq-root*[*of x*] *x-def*(*1*) ‹*p*=*e*› *wf-arcs* **by** *simp*
**then have** *cas-p-r*: *pre-digraph.cas* (*from-dtree dt dh* (*Node r xs*)) (*root x*) *ps' v*
**using** *ind.prems*(*2*) *p-def* **by** *fastforce*
**moreover have** *0*: *root x* $\in$ *dverts x* **using** *dtree.set-sel*(*1*) **by** *blast*
**ultimately have** *cas-ps*: *pre-digraph.cas* (*from-dtree dt dh x*) (*root x*) *ps' v*
**using** *dtree-cas-in-subtree asm x-def*(*1*) *p-def dtree.set-sel*(*1*) **by** *force*
**have** *dhead* (*Node r xs*) *dh e'* = *root x*
**using** *dhead-in-set-eq-root*[*of x*] *x-def* ‹*e'*=*e*› *wf-arcs* **by** *simp*
**then have** *cas-e-r*: *pre-digraph.cas* (*from-dtree dt dh* (*Node r xs*)) (*root x*) *es' v*
**using** *ind.prems*(*3*) *e'-def* **by** *fastforce*
**then have** *pre-digraph.cas* (*from-dtree dt dh x*) (*root x*) *es' v*
**using** *dtree-cas-in-subtree asm x-def*(*1*) *e'-def 0* **by** *force*
**then have** $ps' = es' \vee \neg\ set\ ps' \subseteq darcs\ x \vee \neg\ set\ es' \subseteq darcs\ x$
**using** *ind.IH cas-ps x-def wf-x* **by** *blast*
**moreover have** *set ps'* $\subseteq$ *darcs x*
**using** *cas-darcs-in-subtree cas-p-r x-def*(*1*) *asm p-def 0 set-subset-Cons* **by** *fast*

```
    moreover have set es' ⊆ darcs x
      using cas-darcs-in-subtree cas-e-r x-def(1) asm e'-def 0 set-subset-Cons by
fast
    ultimately have ps' = es' by blast
    then show False using asm p-def e'-def ‹p=e'› by blast
   qed
  qed
qed

lemma dtree-awalk-unique:
  ⟦v ∈ dverts t; pre-digraph.awalk (from-dtree dt dh t) (root t) ps v;
    pre-digraph.awalk (from-dtree dt dh t) (root t) es v⟧
    ⟹ ps = es
  unfolding pre-digraph.awalk-def using cas-unique-in-darcs by fastforce

lemma dtree-unique-awalk-exists:
  assumes v ∈ dverts t
  shows ∃!p. pre-digraph.awalk (from-dtree dt dh t) (root t) p v
  using dtree-awalk-exists dtree-awalk-unique assms by blast

lemma from-dtree-directed: directed-tree (from-dtree dt dh t) (root t)
  apply(unfold-locales)
 by(auto simp: dtail-in-dverts dhead-in-dverts dtree.set-sel(1) dtree-unique-awalk-exists)

theorem from-dtree-fin-directed: finite-directed-tree (from-dtree dt dh t) (root t)
  apply(unfold-locales)
 by(auto simp: dtail-in-dverts dhead-in-dverts dtree.set-sel(1) dtree-unique-awalk-exists
     finite-dverts finite-darcs)
```

### 7.3.3 Identity of Transformation Operations

```
lemma dhead-img-eq-root-img:
  Node r xs = t
    ⟹ (λe. ((dhead (Node r xs) dh e), e)) ‘ snd ‘ fset xs = (λ(x,e). (root x, e)) ‘
fset xs
  using dhead-in-set-eq-root wf-arcs snd-conv image-image disjoint-darcs-if-wf-xs
  by (smt (verit) case-prodE case-prod-conv image-cong)

lemma childarcs-in-out-arcs:
  ⟦Node r xs = t; e ∈ snd ‘ fset xs⟧ ⟹ e ∈ out-arcs (from-dtree dt dh t) r
  by force

lemma out-arcs-in-childarcs:
  assumes Node r xs = t and e ∈ out-arcs (from-dtree dt dh t) r
  shows e ∈ snd ‘ fset xs
proof (rule ccontr)
  assume asm: e ∉ snd ‘ fset xs
  have e ∈ darcs t using assms(2) by simp
  then obtain x e1 where x-def: (x,e1) ∈ fset xs ∧ e ∈ darcs x using assms(1)
```

*asm* **by** *force*
**then have** *dtail t dt e* ∈ *dverts x* **using** *assms*(*1*) *dtail-in-childverts* **by** *blast*
**moreover have** *r* ∉ *dverts x* **using** *assms*(*1*) *wf-verts x-def* **by** (*auto simp*: *wf-dverts-iff-dverts'*)
**ultimately show** *False* **using** *assms*(*2*) **by** *simp*
**qed**

**lemma** *childarcs-eq-out-arcs*:
*Node r xs* = *t* ⟹ *snd* ‘ *fset xs* = *out-arcs* (*from-dtree dt dh t*) *r*
**using** *childarcs-in-out-arcs out-arcs-in-childarcs* **by** *fast*

**lemma** *dtail-in-subtree-eq-subtree*:
⟦*is-subtree t1 t*; *e* ∈ *darcs t1*⟧ ⟹ *dtail t def e* = *dtail t1 def e*
**using** *wf-arcs* **proof**(*induction t rule*: *darcs-mset.induct*)
**case** (*1 r xs*)
**show** *?case*
**proof**(*cases Node r xs*=*t1*)
**case** *False*
**then obtain** *x e1* **where** *x-def*: (*x*,*e1*) ∈ *fset xs* ∧ *is-subtree t1 x* **using** *1.prems*(*1*) **by** *auto*
**then have** *e* ∈ *darcs x* **using** *1.prems*(*2*) *darcs-subtree-subset* **by** *blast*
**then have** *dtail* (*Node r xs*) *def e* = *dtail x def e*
**using** *dtail-in-child-eq-child*[*of x*] *x-def 1.prems*(*3*) **by** *blast*
**then show** *?thesis* **using** *1.IH x-def 1.prems*(*2*−*3*) **by** *fastforce*
**qed** (*simp*)
**qed**

**lemma** *dtail-in-subdverts*:
**assumes** *e* ∈ *darcs x* **and** *is-subtree x t*
**shows** *dtail t def e* ∈ *dverts x*
**proof** −
**interpret** *X*: *wf-dtree x* **by** (*simp add*: *assms*(*2*) *wf-dtree-sub*)
**have** *dtail t def e* = *dtail x def e* **using** *dtail-in-subtree-eq-subtree assms*(*1*,*2*) **by** *blast*
**then show** *?thesis* **using** *assms*(*1*) *X.dtail-in-dverts* **by** *simp*
**qed**

**lemma** *dhead-in-subtree-eq-subtree*:
⟦*is-subtree t1 t*; *e* ∈ *darcs t1*⟧ ⟹ *dhead t def e* = *dhead t1 def e*
**using** *wf-arcs* **proof**(*induction t*)
**case** (*Node r xs*)
**show** *?case*
**proof**(*cases Node r xs*=*t1*)
**case** *False*
**then obtain** *x e1* **where** *x-def*: (*x*,*e1*) ∈ *fset xs* ∧ *is-subtree t1 x* **using** *Node.prems*(*1*) **by** *auto*
**then have** *e* ∈ *darcs x* **using** *Node.prems*(*2*) *darcs-subtree-subset* **by** *blast*
**then have** *dhead* (*Node r xs*) *def e* = *dhead x def e*
**using** *dhead-in-child-eq-child*[*of x*] *x-def Node.prems*(*3*) **by** *force*

```
    then show ?thesis using Node.IH x-def Node.prems(2−3) by fastforce
  qed (simp)
qed

lemma subarcs-in-out-arcs:
  assumes is-subtree (Node r xs) t and e ∈ snd ‘ fset xs
  shows e ∈ out-arcs (from-dtree dt dh t) r
proof −
  have e ∈ darcs (Node r xs) using assms(2) by force
  then have tail (from-dtree dt dh t) e = r
    using dtail-in-subtree-eq-subtree assms(1,2) by auto
  then show ?thesis using darcs-subtree-subset assms(1,2) by fastforce
qed

lemma darc-in-sub-if-dtail-in-sub:
  assumes dtail t dt e = v and e ∈ darcs t and (x,e1) ∈ fset xs
      and is-subtree t1 x and Node r xs = t and v ∈ dverts t1
    shows e ∈ darcs x
proof (rule ccontr)
  assume asm: e ∉ darcs x
  have e ∉ snd ‘ fset xs
    using assms(1−6) asm arc-in-subtree-if-tail-in-subtree dverts-subtree-subset
    by (metis subset-eq)
  then obtain x2 e2 where x2-def: (x2,e2) ∈ fset xs ∧ e ∈ darcs x2 using
assms(2,5) by force
  then have v ∈ dverts x using assms(4,6) dverts-subtree-subset by fastforce
  then have v ∉ dverts x2 using assms(1−3,5) arc-in-subtree-if-tail-in-subtree
asm by blast
  then have dtail x2 dt e ≠ v using assms(1,5) dtail-in-childverts x2-def by fast
  then have dtail t dt e = dtail x2 dt e
    using assms(1,5) x2-def ‹v ∉ dverts x2› dtail-in-childverts by blast
  then show False using assms(1) ‹dtail x2 dt e ≠ v› by simp
qed

lemma out-arcs-in-subarcs-aux:
  assumes is-subtree (Node r xs) t and dtail t dt e = r and e ∈ darcs t
  shows e ∈ snd ‘ fset xs
using assms wf-dtree-axioms proof(induction t)
  case (Node v ys)
  then interpret wf-dtree Node v ys by blast
  show ?case
  proof(cases Node v ys = Node r xs)
    case True
    then show ?thesis using dtail-root-in-set Node.prems(2,3) by blast
  next
    case False
    then obtain x e1 where x-def: (x,e1) ∈ fset ys ∧ is-subtree (Node r xs) x
      using Node.prems(1) by auto
    then have e ∈ darcs x
```

```
    using darc-in-sub-if-dtail-in-sub Node.prems(2,3) dtree.set-intros(1) by fast
   moreover from this have dtail x dt e = r
    using dtail-in-child-eq-child[of x] x-def Node.prems(2) wf-arcs by force
  moreover from this have wf-dtree x using wf-verts wf-arcs x-def by(unfold-locales)
auto
   ultimately show ?thesis using Node.IH x-def by force
 qed
qed

lemma out-arcs-in-subarcs:
 assumes is-subtree (Node r xs) t and e ∈ out-arcs (from-dtree dt dh t) r
 shows e ∈ snd ‘ fset xs
 using assms out-arcs-in-subarcs-aux by auto

lemma subarcs-eq-out-arcs:
 is-subtree (Node r xs) t ⟹ snd ‘ fset xs = out-arcs (from-dtree dt dh t) r
 using subarcs-in-out-arcs out-arcs-in-subarcs by fast

lemma dhead-sub-img-eq-root-img:
 is-subtree (Node v ys) t
  ⟹ (λe. ((dhead t dh e), e)) ‘ snd ‘ fset ys = (λ(x,e). (root x, e)) ‘ fset ys
using wf-dtree-axioms proof(induction t)
 case (Node r xs)
 then interpret wf-dtree Node r xs by blast
 show ?case
 proof(cases Node v ys = Node r xs)
  case True
  then show ?thesis using dhead-img-eq-root-img by simp
 next
  case False
  then obtain x e where x-def: (x,e) ∈ fset xs ∧ is-subtree (Node v ys) x
   using Node.prems(1) by auto
  then interpret X: wf-dtree x using wf-verts wf-arcs by(unfold-locales) auto
  have ∀ a ∈ snd ‘ fset ys. (λe. ((dhead (Node r xs) dh e), e)) a = (λe. ((dhead
x dh e), e)) a
  proof
   fix a
   assume asm: a ∈ snd ‘ fset ys
   then have a ∈ darcs x using x-def darcs-subtree-subset by fastforce
  then show (λe. ((dhead (Node r xs) dh e), e)) a = (λe. ((dhead x dh e), e)) a
    using dhead-in-child-eq-child[of x] x-def wf-arcs by auto
  qed
  then have (λe. ((dhead (Node r xs) dh e), e)) ‘ snd ‘ fset ys
        = (λe. ((dhead x dh e), e)) ‘ snd ‘ fset ys
   by (meson image-cong)
  then show ?thesis using Node.IH x-def X.wf-dtree-axioms by force
 qed
qed
```

**lemma** *subtree-to-dtree-aux-eq*:
**assumes** *is-subtree x t* **and** $v \in$ *dverts x*
**shows** *finite-directed-tree.to-dtree-aux* (*from-dtree dt dh t*) *v*
$=$ *finite-directed-tree.to-dtree-aux* (*from-dtree dt dh x*) *v*
$\wedge$ *finite-directed-tree.to-dtree-aux* (*from-dtree dt dh x*) (*root x*) $=$ *x*
**using** *assms wf-dtree-axioms* **proof**(*induction x arbitrary*: *t v rule*: *darcs-mset.induct*)
**case** *ind*: (*1 r xs*)
**then interpret** *wf-dtree t* **by** *blast*
**obtain** *r' xs'* **where** *r'-def*: *t* $=$ *Node r' xs'* **using** *dtree.exhaust* **by** *auto*
**interpret** *R-xs*: *wf-dtree Node r xs* **using** *ind.prems*(*1*,*3*) *wf-dtree-sub* **by** *simp*
**let** *?todt* $=$ *finite-directed-tree.to-dtree-aux*
**let** *?T* $=$ (*from-dtree dt dh t*)
**let** *?X* $=$ (*from-dtree dt dh* (*Node r xs*))
**interpret** *DT*: *finite-directed-tree ?T root t* **using** *from-dtree-fin-directed* **by** *blast*
**interpret** *XT*: *finite-directed-tree ?X root* (*Node r xs*)
**using** *R-xs.from-dtree-fin-directed* **by** *blast*

**have** *ih*: $\forall$ *y* $\in$ *fset xs*. ($\lambda$(*x*,*e*). (*XT.to-dtree-aux* (*root x*), *e*)) *y* $=$ *y*
**proof**
**fix** *y*
**assume** *asm*: *y* $\in$ *fset xs*
**obtain** *x e* **where** *x-def*: *y* $=$ (*x*,*e*) **by** *fastforce*
**then have** *is-subtree x* (*Node r xs*) **using** *subtree-if-child asm*
**by** (*metis image-iff prod.sel*(*1*))
**then have** *?todt* (*from-dtree dt dh x*) (*root x*) $=$ *x*
$\wedge$ *XT.to-dtree-aux* (*root x*) $=$ *?todt* (*from-dtree dt dh x*) (*root x*)
**using** *ind.IH R-xs.wf-dtree-axioms asm x-def dtree.set-sel*(*1*) **by** *blast*
**then have** *XT.to-dtree-aux* (*root x*) $=$ *x* **by** *simp*
**then show** ($\lambda$(*x*,*e*). (*XT.to-dtree-aux* (*root x*), *e*)) *y* $=$ *y* **using** *x-def* **by** *fast*
**qed**
**let** *?f* $=$ $\lambda$(*x*,*e*). (*XT.to-dtree-aux x*, *e*)
**let** *?g* $=$ $\lambda$*e*. ((*dhead* (*Node r xs*) *dh e*), *e*)
**obtain** *ys* **where** *ys-def*: *XT.to-dtree-aux* (*root* (*Node r xs*)) $=$ *Node r ys*
**using** *dtree.exhaust dtree.sel*(*1*) *XT.to-dtree-aux-root* **by** *metis*
**then have** *fset ys* $=$ ($\lambda$*e*. (*XT.to-dtree-aux* (*head ?X e*), *e*)) ‘ *out-arcs ?X r*
**using** *XT.dtree-children-img-alt XT.dtree-children-fset-id dtree.sel*(*1*) **by** (*smt* (*verit*))
**also have** ... $=$ ($\lambda$*e*. (*XT.to-dtree-aux* (*dhead* (*Node r xs*) *dh e*), *e*)) ‘ (*snd* ‘ *fset xs*)
**using** *R-xs.childarcs-eq-out-arcs* **by** *simp*
**also have** ... $=$ *?f* ‘ *?g* ‘ (*snd* ‘ *fset xs*) **by** *fast*
**also have** ... $=$ *?f* ‘ ($\lambda$(*x*,*e*). (*root x*, *e*)) ‘ *fset xs* **using** *R-xs.dhead-img-eq-root-img* **by** *simp*
**also have** ... $=$ ($\lambda$(*x*,*e*). (*XT.to-dtree-aux* (*root x*), *e*)) ‘ *fset xs* **by** *fast*
**also have** ... $=$ *fset xs* **using** *ih* **by** *simp*
**finally have** *g2*: *ys* $=$ *xs* **by** (*simp add*: *fset-inject*)

**show** *?case*

```
 proof(cases v = r)
  case True

  have 0: ∀ y ∈ fset xs. (λ(x,e). (DT.to-dtree-aux (root x), e)) y = y
  proof
   fix y
   assume asm: y ∈ fset xs
   obtain x e where x-def: y = (x,e) by fastforce
   then have is-subtree x (Node r xs) using subtree-if-child asm
    by (metis image-iff prod.sel(1))
   then have is-subtree x t using asm subtree-trans ind.prems(1) by blast
   then have ?todt (from-dtree dt dh x) (root x) = x
         ∧ DT.to-dtree-aux (root x) = ?todt (from-dtree dt dh x) (root x)
    using ind.IH wf-dtree-axioms asm x-def dtree.set-sel(1) by blast
   then have DT.to-dtree-aux (root x) = x by simp
   then show (λ(x,e). (DT.to-dtree-aux (root x), e)) y = y using x-def by fast
  qed
  let ?f = λ(x,e). (DT.to-dtree-aux x, e)
  let ?g = λe. ((dhead (Node r' xs') dh e), e)
  obtain zs where zs-def: DT.to-dtree-aux v = Node v zs
   using dtree.exhaust by simp
  then have fset zs = (λe. (DT.to-dtree-aux (head ?T e), e)) ‘ out-arcs ?T r
   using DT.dtree-children-img-alt DT.dtree-children-fset-id True by presburger
  also have ... = (λe. (DT.to-dtree-aux (dhead t dh e), e)) ‘ (snd ‘ fset xs)
   using ind.prems(1) subarcs-eq-out-arcs by force
  also have ... = ?f ‘ ?g ‘ (snd ‘ fset xs) using r'-def by fast
  also have ... = ?f ‘ (λ(x,e). (root x, e)) ‘ fset xs
   using dhead-sub-img-eq-root-img ind.prems(1) r'-def by blast
  also have ... = (λ(x,e). (DT.to-dtree-aux (root x), e)) ‘ fset xs by fast
  also have ... = fset xs using 0 by simp
  finally have g1: zs = xs by (simp add: fset-inject)
  then show ?thesis using zs-def True g2 ys-def by simp
 next
  case False

   then obtain x1 e1 where x-def: (x1,e1) ∈ fset xs v ∈ dverts x1 using
ind.prems(2) by auto
  then have is-subtree x1 (Node r xs) using subtree-if-child
   by (metis image-iff prod.sel(1))
  moreover from this have is-subtree x1 t using ind.prems(1) subtree-trans by
blast
  ultimately have g1: DT.to-dtree-aux v = XT.to-dtree-aux v
   using ind.IH x-def by (metis R-xs.wf-dtree-axioms wf-dtree-axioms)
  then show ?thesis using g1 g2 ys-def by blast
 qed
qed

interpretation T: finite-directed-tree from-dtree dt dh t root t
 using from-dtree-fin-directed by simp
```

**lemma** *to-from-dtree-aux-id*: *T.to-dtree-aux dt dh* (*root t*) = *t*
**using** *subtree-to-dtree-aux-eq dtree.set-sel*(*1*) *self-subtree* **by** *blast*

**theorem** *to-from-dtree-id*: *T.to-dtree dt dh* = *t*
**using** *to-from-dtree-aux-id T.to-dtree-def* **by** *simp*

**end**

**context** *finite-directed-tree*
**begin**

**lemma** *wf-to-dtree-aux*: *wf-dtree* (*to-dtree-aux r*)
**unfolding** *wf-dtree-def* **using** *wf-dverts-to-dtree-aux wf-darcs-to-dtree-aux* **by** *blast*

**theorem** *wf-to-dtree*: *wf-dtree to-dtree*
**unfolding** *to-dtree-def* **using** *wf-to-dtree-aux* **by** *blast*

**end**

## 7.4 Degrees of Nodes

**fun** *max-deg* :: ('*a*,'*b*) *dtree* ⇒ *nat* **where**
*max-deg* (*Node r xs*) = (*if xs* = {||} *then 0 else max* (*Max* (*max-deg* ‘ *fst* ‘ *fset xs*)) (*fcard xs*))

**lemma** *mdeg-eq-fcard-if-empty*: *xs* = {||} ⟹ *max-deg* (*Node r xs*) = *fcard xs*
**by** *simp*

**lemma** *mdeg0-if-fcard0*: *fcard xs* = *0* ⟹ *max-deg* (*Node r xs*) = *0*
**by** *simp*

**lemma** *mdeg0-iff-fcard0*: *fcard xs* = *0* ⟷ *max-deg* (*Node r xs*) = *0*
**by** *simp*

**lemma** *nempty-if-mdeg-gt-fcard*: *max-deg* (*Node r xs*) > *fcard xs* ⟹ *xs* ≠ {||}
**by** *auto*

**lemma** *mdeg-img-nempty*: *max-deg* (*Node r xs*) > *fcard xs* ⟹ *max-deg* ‘ *fst* ‘ *fset xs* ≠ {}
**using** *nempty-if-mdeg-gt-fcard*[*of xs*] **by** *fast*

**lemma** *mdeg-img-fin*: *finite* (*max-deg* ‘ *fst* ‘ *fset xs*)
**by** *simp*

**lemma** *mdeg-Max-if-gt-fcard*:
*max-deg* (*Node r xs*) > *fcard xs* ⟹ *max-deg* (*Node r xs*) = *Max* (*max-deg* ‘ *fst* ‘ *fset xs*)

**by** (*auto split*: *if-splits*)

**lemma** *mdeg-child-if-gt-fcard*:
*max-deg* (*Node r xs*) > *fcard xs* ⟹ ∃ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* = *max-deg* (*Node r xs*)
**unfolding** *mdeg-Max-if-gt-fcard* **using** *Max-in*[*OF mdeg-img-fin mdeg-img-nempty*]
**by** *force*

**lemma** *mdeg-child-if-wedge*:
⟦*max-deg* (*Node r xs*) > *n*; *fcard xs* ≤ *n* ∨ ¬(∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *n*)⟧
⟹ ∃ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* > *n*
**using** *mdeg-child-if-gt-fcard*[*of xs*] **by** *force*

**lemma** *maxif-eq-Max*: *finite X* ⟹ (*if X* ≠ {} *then max x* (*Max X*) *else x*) = *Max* (*insert x X*)
**by** *simp*

**lemma** *mdeg-img-empty-iff*: *max-deg* ‘ *fst* ‘ *fset xs* = {} ⟷ *xs* = {||}
**by** *fast*

**lemma** *mdeg-alt*: *max-deg* (*Node r xs*) = *Max* (*insert* (*fcard xs*) (*max-deg* ‘ *fst* ‘ *fset xs*))
**using** *maxif-eq-Max*[*OF mdeg-img-fin*, *of xs fcard xs*] *mdeg-img-empty-iff*[*of xs*]
**by** (*auto split*: *if-splits*)

**lemma** *finite-fMax-union*: *finite Y* ⟹ *finite* (⋃ *y*∈*Y*. {*Max* (*f y*)})
**by** *blast*

**lemma** *Max-union-Max-out*:
**assumes** *finite Y* **and** ∀ *y* ∈ *Y*. *finite* (*f y*) **and** ∀ *y* ∈ *Y*. *f y* ≠ {} **and** *Y* ≠ {}
**shows** *Max* (⋃ *y*∈*Y*. {*Max* (*f y*)}) = *Max* (⋃ *y*∈*Y*. *f y*) (**is** *?M1*=-)
**proof** −
**have** ∀ *y* ∈ *Y*. ∀ *x* ∈ *f y*. *Max* (*f y*) ≥ *x* **using** *assms*(*2*) **by** *simp*
**moreover have** ∀ *x* ∈ (⋃ *y*∈*Y*. {*Max* (*f y*)}). *?M1* ≥ *x* **using** *assms*(*1*) **by** *simp*
**moreover have** *M1-in*: *?M1* ∈ (⋃ *y*∈*Y*. {*Max* (*f y*)})
**using** *assms*(*1*,*4*) *Max-in*[*OF finite-fMax-union*] **by** *auto*
**ultimately have** ∀ *y* ∈ *Y*. ∀ *x* ∈ *f y*. *?M1* ≥ *x* **by** *force*
**then have** ∀ *x* ∈ (⋃ *y*∈*Y*. *f y*). *?M1* ≥ *x* **by** *blast*
**moreover have** *?M1* ∈ (⋃ *y*∈*Y*. (*f y*)) **using** *M1-in assms*(*2*−*4*) **by** *force*
**ultimately show** *?thesis* **using** *assms*(*1*,*2*) *Max-eqI finite-UN-I* **by** *metis*
**qed**

**lemma** *Max-union-Max-out-insert*:
⟦*finite Y*; ∀ *y* ∈ *Y*. *finite* (*f y*); ∀ *y* ∈ *Y*. *f y* ≠ {}; *Y* ≠ {}⟧
⟹ *Max* (*insert x* (⋃ *y*∈*Y*. {*Max* (*f y*)})) = *Max* (*insert x* (⋃ *y*∈*Y*. *f y*))
**using** *Max-union-Max-out*[*of Y f*] **by** *simp*

**lemma** *mdeg-alt2*: *max-deg t* = *Max* {*fcard* (*sucs x*)|*x*. *is-subtree x t*}
**proof**(*induction t rule*: *max-deg.induct*)

**case** (*1 r xs*)
**then show** *?case*
**proof**(*cases xs* = {||})
**case** *False*
**let** *?f* = (λ*t1*. {*fcard* (*sucs x*)|*x*. *is-subtree x t1*})
**let** *?f′* = (λ*t1*. (λ*x*. *fcard* (*sucs x*)) ‘ {*x*. *is-subtree x t1*})
**have** *fin*: *finite* (*fst* ‘ *fset xs*) **by** *simp*
**have** *f-eq1*: *?f* = *?f′* **by** *blast*
**then have** *f-eq*: ∀ *y*∈*fst* ‘ *fset xs*. (*?f y* = *?f′ y*) **by** *blast*
**moreover have** ∀ *y*∈*fst* ‘ *fset xs*. *finite* (*?f′ y*) **using** *finite-subtrees* **by** *blast*
**ultimately have** *fin′*: ∀ *y*∈*fst* ‘ *fset xs*. *finite* (*?f y*) **by** *simp*
**have** *nempty*: ∀ *y*∈*fst* ‘ *fset xs*. {*fcard* (*sucs x*) |*x*. *is-subtree x y*} ≠ {}
**using** *self-subtree* **by** *blast*
**have** *max-deg* ‘ *fst* ‘ *fset xs* = (⋃ *t1*∈*fst* ‘ *fset xs*. {*Max* (*?f t1*)})
**using** *1.IH*[*OF False*] **by** *auto*
**then have** *max-deg* (*Node r xs*) = *Max* (*insert* (*fcard xs*) (⋃ *t1*∈*fst* ‘ *fset xs*. {*Max* (*?f t1*)}))
**using** *mdeg-alt*[*of r xs*] **by** *simp*
**also have** ... = *Max* (*insert* (*fcard xs*) (⋃ *t1*∈*fst* ‘ *fset xs*. *?f t1*))
**using** *Max-union-Max-out-insert*[*OF fin fin′ nempty*] **by** *fastforce*
**also have** ... = *Max* (*insert* (*fcard xs*) ((⋃ *t1*∈*fst* ‘ *fset xs*. *?f′ t1*)))
**using** *f-eq* **by** *simp*
**also have** ...
= *Max* (*insert* (*fcard xs*) ((⋃ *t1*∈*fst* ‘ *fset xs*. *fcard* ‘ *sucs* ‘ {*x*. *is-subtree x t1*})))
**using** *image-image* **by** *metis*
**also have** ...
= *Max* (*insert* (*fcard xs*) (*fcard* ‘ *sucs* ‘ (⋃ *t1*∈*fst* ‘ *fset xs*. {*x*. *is-subtree x t1*})))
**by** (*metis image-UN*)
**also have** ...
= *Max* (*fcard* ‘ *sucs* ‘ (*insert* (*Node r xs*) (⋃ *t1*∈*fst* ‘ *fset xs*. {*x*. *is-subtree x t1*})))
**by** *force*
**also have** ... = *Max* (*fcard* ‘ *sucs* ‘ {*x*. *is-subtree x* (*Node r xs*)})
**unfolding** *subtrees-insert-union* **by** *blast*
**finally show** *?thesis* **using** *f-eq1 image-image* **by** *metis*
**qed**(*simp*)
**qed**

**lemma** *mdeg-singleton*: *max-deg* (*Node r* {|(*t1*,*e1*)|}) = *max* (*max-deg t1*) (*fcard* {|(*t1*,*e1*)|})
**by** *simp*

**lemma** *mdeg-ge-child-aux*: (*t1*,*e1*) ∈ *fset xs* ⟹ *max-deg t1* ≤ *Max* (*max-deg* ‘ *fst* ‘ *fset xs*)
**using** *Max-ge*[*OF mdeg-img-fin*] **by** *fastforce*

**lemma** *mdeg-ge-child*: (*t1*,*e1*) ∈ *fset xs* ⟹ *max-deg t1* ≤ *max-deg* (*Node r xs*)

**using** *mdeg-ge-child-aux* **by** *fastforce*

**lemma** *mdeg-ge-child'*: *t1* ∈ *fst* ‘ *fset xs* ⟹ *max-deg t1* ≤ *max-deg* (*Node r xs*)
**using** *mdeg-ge-child*[*of t1*] **by** *force*

**lemma** *mdeg-ge-sub*: *is-subtree t1 t2* ⟹ *max-deg t1* ≤ *max-deg t2*
**proof**(*induction t2*)
**case** (*Node r xs*)
**show** *?case*
**proof**(*cases Node r xs=t1*)
**case** *False*
**then obtain** *x e1* **where** *x-def*: (*x*,*e1*) ∈ *fset xs is-subtree t1 x* **using** *Node.prems*(*1*)
**by** *auto*
**then have** *max-deg t1* ≤ *max-deg x* **using** *Node.IH* **by** *force*
**then show** *?thesis* **using** *mdeg-ge-child*[*OF x-def*(*1*)] **by** *simp*
**qed**(*simp*)
**qed**

**lemma** *mdeg-gt-0-if-nempty*: *xs* ≠ {||} ⟹ *max-deg* (*Node r xs*) > *0*
**using** *fcard-fempty* **by** *auto*

**corollary** *empty-if-mdeg-0*: *max-deg* (*Node r xs*) = *0* ⟹ *xs* = {||}
**using** *mdeg-gt-0-if-nempty* **by** (*metis less-numeral-extra*(*3*))

**lemma** *nempty-if-mdeg-n0*: *max-deg* (*Node r xs*) ≠ *0* ⟹ *xs* ≠ {||}
**by** *auto*

**corollary** *empty-iff-mdeg-0*: *max-deg* (*Node r xs*) = *0* ⟷ *xs* = {||}
**using** *nempty-if-mdeg-n0 empty-if-mdeg-0* **by** *auto*

**lemma** *mdeg-root*: *max-deg* (*Node r xs*) = *max-deg* (*Node v xs*)
**by** *simp*

**lemma** *mdeg-ge-fcard*: *fcard xs* ≤ *max-deg* (*Node r xs*)
**by** *simp*

**lemma** *mdeg-fcard-if-fcard-ge-child*:
∀ (*t*,*e*) ∈ *fset xs*. *max-deg t* ≤ *fcard xs* ⟹ *max-deg* (*Node r xs*) = *fcard xs*
**using** *mdeg-child-if-gt-fcard*[*of xs r*] *mdeg-ge-fcard*[*of xs r*] **by** *fastforce*

**lemma** *mdeg-fcard-if-fcard-ge-child'*:
∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *fcard xs* ⟹ *max-deg* (*Node r xs*) = *fcard xs*
**using** *mdeg-fcard-if-fcard-ge-child*[*of xs r*] **by** *fastforce*

**lemma** *fcard-single-1*: *fcard* {|*x*|} = *1*
**by** (*simp add*: *fcard-finsert*)

**lemma** *fcard-single-1-iff*: *fcard xs* = *1* ⟷ (∃ *x*. *xs* = {|*x*|})
**by** (*metis all-not-fin-conv bot.extremum fcard-seteq fcard-single-1*

*finsert-fsubset le-numeral-extra*(*4*))

**lemma** *fcard-not0-if-elem*: $\exists\, x.\ x \in \mathit{fset}\ \mathit{xs} \Longrightarrow \mathit{fcard}\ \mathit{xs} \neq 0$
**by** *auto*

**lemma** *fcard1-if-le1-elem*: $[\![\mathit{fcard}\ \mathit{xs} \leq 1;\ x \in \mathit{fset}\ \mathit{xs}]\!] \Longrightarrow \mathit{fcard}\ \mathit{xs} = 1$
**using** *fcard-not0-if-elem*[*of xs*] **by** *fastforce*

**lemma** *singleton-if-fcard-le1-elem*: $[\![\mathit{fcard}\ \mathit{xs} \leq 1;\ x \in \mathit{fset}\ \mathit{xs}]\!] \Longrightarrow \mathit{xs} = \{|x|\}$
**using** *fcard-single-1-iff*[*of xs*] *fcard1-if-le1-elem* **by** *fastforce*

**lemma** *singleton-if-mdeg-le1-elem*: $[\![\mathit{max\text{-}deg}\ (\mathit{Node}\ r\ \mathit{xs}) \leq 1;\ x \in \mathit{fset}\ \mathit{xs}]\!] \Longrightarrow \mathit{xs} = \{|x|\}$
**using** *singleton-if-fcard-le1-elem*[*of xs*] *mdeg-ge-fcard*[*of xs*] **by** *simp*

**lemma** *singleton-if-mdeg-le1-elem-suc*: $[\![\mathit{max\text{-}deg}\ t \leq 1;\ x \in \mathit{fset}\ (\mathit{sucs}\ t)]\!] \Longrightarrow \mathit{sucs}\ t = \{|x|\}$
**using** *singleton-if-mdeg-le1-elem*[*of root t sucs t*] **by** *simp*

**lemma** *fcard0-if-le1-not-singleton*: $[\![\forall\, x.\ \mathit{xs} \neq \{|x|\};\ \mathit{fcard}\ \mathit{xs} \leq 1]\!] \Longrightarrow \mathit{fcard}\ \mathit{xs} = 0$
**using** *fcard-single-1-iff*[*of xs*] **by** *fastforce*

**lemma** *empty-fset-if-fcard-le1-not-singleton*: $[\![\forall\, x.\ \mathit{xs} \neq \{|x|\};\ \mathit{fcard}\ \mathit{xs} \leq 1]\!] \Longrightarrow \mathit{xs} = \{||\}$
**using** *fcard0-if-le1-not-singleton* **by** *auto*

**lemma** *fcard0-if-mdeg-le1-not-single*: $[\![\forall\, x.\ \mathit{xs} \neq \{|x|\};\ \mathit{max\text{-}deg}\ (\mathit{Node}\ r\ \mathit{xs}) \leq 1]\!] \Longrightarrow \mathit{fcard}\ \mathit{xs} = 0$
**using** *fcard0-if-le1-not-singleton*[*of xs*] *mdeg-ge-fcard*[*of xs*] **by** *simp*

**lemma** *empty-fset-if-mdeg-le1-not-single*: $[\![\forall\, x.\ \mathit{xs} \neq \{|x|\};\ \mathit{max\text{-}deg}\ (\mathit{Node}\ r\ \mathit{xs}) \leq 1]\!] \Longrightarrow \mathit{xs} = \{||\}$
**using** *fcard0-if-mdeg-le1-not-single* **by** *auto*

**lemma** *fcard0-if-mdeg-le1-not-single-suc*:
$[\![\forall\, x.\ \mathit{sucs}\ t \neq \{|x|\};\ \mathit{max\text{-}deg}\ t \leq 1]\!] \Longrightarrow \mathit{fcard}\ (\mathit{sucs}\ t) = 0$
**using** *fcard0-if-mdeg-le1-not-single*[*of sucs t root t*] **by** *simp*

**lemma** *empty-fset-if-mdeg-le1-not-single-suc*: $[\![\forall\, x.\ \mathit{sucs}\ t \neq \{|x|\};\ \mathit{max\text{-}deg}\ t \leq 1]\!] \Longrightarrow \mathit{sucs}\ t = \{||\}$
**using** *fcard0-if-mdeg-le1-not-single-suc* **by** *auto*

**lemma** *mdeg-1-singleton*:
**assumes** $\mathit{max\text{-}deg}\ (\mathit{Node}\ r\ \mathit{xs}) = 1$
**shows** $\exists\, x.\ \mathit{xs} = \{|x|\}$
**proof** −
**obtain** *x* **where** *x-def*: $x\ |\in|\ \mathit{xs}$
**using** *assms* **by** (*metis all-not-fin-conv empty-iff-mdeg-0 zero-neq-one*)

**moreover have** *fcard xs* $\leq$ *1* **using** *assms mdeg-ge-fcard* **by** *metis*
**ultimately have** *xs* = {|*x*|}
**by** (*metis order-bot-class.bot.not-eq-extremum diff-Suc-1 diff-is-0-eq' fcard-finsert-disjoint less-nat-zero-code mk-disjoint-finsert pfsubset-fcard-mono*)
**then show** *?thesis* **by** *simp*
**qed**

**lemma** *subtree-child-if-dvert-notr-mdeg-le1*:
**assumes** *max-deg* (*Node r xs*) $\leq$ *1* **and** *v* $\neq$ *r* **and** *v* $\in$ *dverts* (*Node r xs*)
**shows** $\exists$ *r' e zs. is-subtree* (*Node r'* {|(*Node v zs*,*e*)|}) (*Node r xs*)
**proof** −
**obtain** *r' ys zs* **where** *zs-def*: *is-subtree* (*Node r' ys*) (*Node r xs*) *Node v zs* $\in$ *fst* ‘ *fset ys*
**using** *subtree-child-if-dvert-notroot*[*OF assms*(*2*,*3*)] **by** *blast*
**have** *0*: *max-deg* (*Node r' ys*) $\leq$ *1* **using** *mdeg-ge-sub*[*OF zs-def*(*1*)] *assms*(*1*) **by** *simp*
**obtain** *e* **where** {|(*Node v zs*,*e*)|} = *ys*
**using** *singleton-if-mdeg-le1-elem*[*OF 0*] *zs-def*(*2*) **by** *fastforce*
**then show** *?thesis* **using** *zs-def*(*1*) **by** *blast*
**qed**

**lemma** *subtree-child-if-dvert-notroot-mdeg-le1*:
⟦*max-deg t* $\leq$ *1*; *v* $\neq$ *root t*; *v* $\in$ *dverts t*⟧
$\Longrightarrow$ $\exists$ *r' e zs. is-subtree* (*Node r'* {|(*Node v zs*,*e*)|}) *t*
**using** *subtree-child-if-dvert-notr-mdeg-le1*[*of root t sucs t*] **by** *simp*

**lemma** *mdeg-child-sucs-eq-if-gt1*:
**assumes** *max-deg* (*Node r* {|(*t*,*e*)|}) > *1*
**shows** *max-deg* (*Node r* {|(*t*,*e*)|}) = *max-deg* (*Node v* (*sucs t*))
**proof** −
**have** *fcard* {|(*t*,*e*)|} = *1* **using** *fcard-single-1* **by** *fast*
**then have** *max-deg* (*Node r* {|(*t*,*e*)|}) = *max-deg t* **using** *assms* **by** *simp*
**then show** *?thesis* **using** *mdeg-root*[*of root t sucs t v*] *dtree.exhaust-sel*[*of t*] **by** *argo*
**qed**

**lemma** *mdeg-child-sucs-le*: *max-deg* (*Node v* (*sucs t*)) $\leq$ *max-deg* (*Node r* {|(*t*,*e*)|})
**using** *mdeg-root*[*of v sucs t root t*] **by** *simp*

**lemma** *mdeg-eq-child-if-singleton-gt1*:
*max-deg* (*Node r* {|(*t1*,*e1*)|}) > *1* $\Longrightarrow$ *max-deg* (*Node r* {|(*t1*,*e1*)|}) = *max-deg t1*
**using** *mdeg-singleton*[*of r t1*] **by** (*auto simp*: *fcard-single-1 max-def*)

**lemma** *fcard-gt1-if-mdeg-gt-child*:
**assumes** *max-deg* (*Node r xs*) > *n* **and** *t1* $\in$ *fst* ‘ *fset xs* **and** *max-deg t1* $\leq$ *n* **and** *n*$\neq$*0*
**shows** *fcard xs* > *1*
**proof**(*rule ccontr*)

**assume** $\neg$*fcard xs* > *1*
**then have** *fcard xs* $\leq$ *1* **by** *simp*
**then have** $\exists$ *e1*. *xs* = {|(*t1*,*e1*)|} **using** *assms*(*2*) *singleton-if-fcard-le1-elem* **by** *fastforce*
**then show** *False* **using** *mdeg-singleton*[*of r t1*] *assms*(*1*,*3*,*4*) **by** (*auto simp*: *fcard-single-1*)
**qed**

**lemma** *fcard-gt1-if-mdeg-gt-suc*:
⟦*max-deg t2* > *n*; *t1* ∈ *fst* ‘ *fset* (*sucs t2*); *max-deg t1* ≤ *n*; *n*≠*0*⟧ ⟹ *fcard* (*sucs t2*) > *1*
**using** *fcard-gt1-if-mdeg-gt-child*[*of n root t2 sucs t2*] **by** *simp*

**lemma** *fcard-gt1-if-mdeg-gt-child1*:
⟦*max-deg* (*Node r xs*) > *1*; *t1* ∈ *fst* ‘ *fset xs*; *max-deg t1* ≤ *1*⟧ ⟹ *fcard xs* > *1*
**using** *fcard-gt1-if-mdeg-gt-child* **by** *auto*

**lemma** *fcard-gt1-if-mdeg-gt-suc1*:
⟦*max-deg t2* > *1*; *t1* ∈ *fst* ‘ *fset* (*sucs t2*); *max-deg t1* ≤ *1*⟧ ⟹ *fcard* (*sucs t2*) > *1*
**using** *fcard-gt1-if-mdeg-gt-suc* **by** *blast*

**lemma** *fcard-lt-non-inj-f*:
⟦*f a* = *f b*; *a* ∈ *fset xs*; *b* ∈ *fset xs*; *a*≠*b*⟧ ⟹ *fcard* (*f* |‘| *xs*) < *fcard xs*
**proof**(*induction xs*)
**case** (*insert x xs*)
**then consider** *a* ∈ *fset xs b* ∈ *fset xs* | *a* = *x b* ∈ *fset xs* | *a* ∈ *fset xs b* = *x*
**by** *auto*
**then show** *?case*
**proof**(*cases*)
**case** *1*
**then show** *?thesis*
**using** *insert.IH insert.prems*(*1*,*4*) **by** (*simp add*: *fcard-finsert-if*)
**next**
**case** *2*
**then show** *?thesis*
**proof**(*cases fcard* (*f* |‘| *xs*) = *fcard xs*)
**case** *True*
**then show** *?thesis*
**using** *2 insert.hyps insert.prems*(*1*)
**by** (*metis fcard-finsert-disjoint fimage-finsert finsert-fimage lessI*)
**next**
**case** *False*
**then have** *fcard* (*f* |‘| *xs*) ≤ *fcard xs* **using** *fcard-image-le* **by** *auto*
**then have** *fcard* (*f* |‘| *xs*) < *fcard xs* **using** *False* **by** *simp*
**then show** *?thesis*
**using** *2 insert.prems*(*1*) *fcard-image-le fcard-mono fimage-finsert less-le-not-le*
**by** (*metis order-class.order.not-eq-order-implies-strict finsert-fimage fsubset-finsertI*)

```
    qed
  next
    case 3
    then show ?thesis
    proof(cases fcard (f |‘| xs) = fcard xs)
      case True
      then show ?thesis
        using 3 insert.hyps insert.prems(1)
        by (metis fcard-finsert-disjoint fimage-finsert finsert-fimage lessI)
    next
      case False
      then have fcard (f |‘| xs) ≤ fcard xs using fcard-image-le by auto
      then have fcard (f |‘| xs) < fcard xs using False by simp
      then show ?thesis
      using 3 insert.prems(1) fcard-image-le fcard-mono fimage-finsert less-le-not-le
         by (metis order-class.order.not-eq-order-implies-strict finsert-fimage fsub-
set-finsertI)
    qed
  qed
qed (simp)

lemma mdeg-img-le:
  assumes ∀(t,e) ∈ fset xs. max-deg (fst (f (t,e))) ≤ max-deg t
  shows max-deg (Node r (f |‘| xs)) ≤ max-deg (Node r xs)
proof(cases max-deg (Node r (f |‘| xs)) = fcard (f |‘| xs))
  case True
  then show ?thesis using fcard-image-le[of f xs] by auto
next
  case False
  then have max-deg (Node r (f |‘| xs)) > fcard (f |‘| xs)
    using mdeg-ge-fcard[of f |‘| xs] by simp
  then obtain t1 e1 where t1-def:
     (t1,e1) ∈ fset (f |‘| xs) max-deg t1 = max-deg (Node r (f |‘| xs))
    using mdeg-child-if-gt-fcard[of f |‘| xs r]
    by (metis (no-types, opaque-lifting) fst-conv imageE surj-pair)
  then obtain t2 e2 where t2-def: (t2,e2) ∈ fset xs f (t2,e2) = (t1,e1) by auto
  then have max-deg t2 ≥ max-deg (Node r (f |‘| xs)) using t1-def(2) assms by
fastforce
  then show ?thesis using mdeg-ge-child[OF t2-def(1)] by simp
qed

lemma mdeg-img-le′:
  assumes ∀(t,e) ∈ fset xs. max-deg (f t) ≤ max-deg t
  shows max-deg (Node r ((λ(t,e). (f t, e)) |‘| xs)) ≤ max-deg (Node r xs)
  using mdeg-img-le[of xs λ(t,e). (f t, e)] assms by simp

lemma mdeg-le-if-fcard-and-child-le:
  ⟦∀(t,e) ∈ fset xs. max-deg t ≤ m; fcard xs ≤ m⟧ ⟹ max-deg (Node r xs) ≤ m
  using mdeg-ge-fcard mdeg-child-if-gt-fcard[of xs r] by fastforce
```

**lemma** *mdeg-child-if-child-max*:
⟦∀ (*t*,*e*) ∈ *fset xs*. *max-deg t* ≤ *max-deg t1*; *fcard xs* ≤ *max-deg t1*; (*t1*,*e1*) ∈ *fset xs*⟧
⟹ *max-deg* (*Node r xs*) = *max-deg t1*
**using** *mdeg-le-if-fcard-and-child-le*[*of xs max-deg t1*] *mdeg-ge-child*[*of t1 e1 xs*]
**by** *simp*

**corollary** *mdeg-child-if-child-max′*:
⟦∀ (*t*,*e*) ∈ *fset xs*. *max-deg t* ≤ *max-deg t1*; *fcard xs* ≤ *max-deg t1*; *t1* ∈ *fst* ‘ *fset xs*⟧
⟹ *max-deg* (*Node r xs*) = *max-deg t1*
**using** *mdeg-child-if-child-max*[*of xs t1*] **by** *force*

**lemma** *mdeg-img-eq*:
**assumes** ∀ (*t*,*e*) ∈ *fset xs*. *max-deg* (*fst* (*f* (*t*,*e*))) = *max-deg t*
**and** *fcard* (*f* |‘| *xs*) = *fcard xs*
**shows** *max-deg* (*Node r* (*f* |‘| *xs*)) = *max-deg* (*Node r xs*)
**proof**(*cases max-deg* (*Node r* (*f* |‘| *xs*)) = *fcard* (*f* |‘| *xs*))
**case** *True*
**then have** ∀ (*t*,*e*) ∈ *fset* (*f* |‘| *xs*). *max-deg t* ≤ *fcard* (*f* |‘| *xs*)
**using** *mdeg-ge-child*
**by** (*metis* (*mono-tags*, *lifting*) *case-prodI2*)
**then have** ∀ (*t*,*e*) ∈ *fset xs*. *max-deg t* ≤ *fcard xs* **using** *assms* **by** *fastforce*
**then have** *max-deg* (*Node r xs*) = *fcard xs* **using** *mdeg-fcard-if-fcard-ge-child* **by** *fast*
**then show** *?thesis* **using** *True assms*(*2*) **by** *simp*
**next**
**case** *False*
**then have** *max-deg* (*Node r* (*f* |‘| *xs*)) > *fcard* (*f* |‘| *xs*)
**using** *mdeg-ge-fcard*[*of f* |‘| *xs*] **by** *simp*
**then obtain** *t1 e1* **where** *t1-def*:
(*t1*,*e1*) ∈ *fset* (*f* |‘| *xs*) *max-deg t1* = *max-deg* (*Node r* (*f* |‘| *xs*))
**using** *mdeg-child-if-gt-fcard*[*of f* |‘| *xs r*]
**by** (*metis* (*no-types*, *opaque-lifting*) *fst-conv imageE old.prod.exhaust*)
**then obtain** *t2 e2* **where** *t2-def*: (*t2*,*e2*) ∈ *fset xs f* (*t2*,*e2*) = (*t1*,*e1*) **by** *auto*
**then have** *mdeg-t21*: *max-deg t2* = *max-deg t1* **using** *assms*(*1*) **by** *auto*
**have** ∀ (*t3*,*e3*) ∈ *fset* (*f* |‘| *xs*). *max-deg t3* ≤ *max-deg t1*
**using** *t1-def*(*2*) *mdeg-ge-child*[**where** *xs*=*f* |‘| *xs*]
**by** (*metis* (*no-types*, *lifting*) *case-prodI2*)
**then have** ∀ (*t3*,*e3*) ∈ *fset xs*. *max-deg* (*fst* (*f* (*t3*,*e3*))) ≤ *max-deg t1* **by** *auto*
**then have** ∀ (*t3*,*e3*) ∈ *fset xs*. *max-deg t3* ≤ *max-deg t2* **using** *assms*(*1*) *mdeg-t21* **by** *fastforce*
**moreover have** *max-deg t2* ≥ *fcard xs* **using** *t1-def*(*2*) *assms*(*2*) *mdeg-t21* **by** *simp*
**ultimately have** *max-deg* (*Node r xs*) = *max-deg t2*
**using** *t2-def*(*1*) *mdeg-child-if-child-max* **by** *metis*
**then show** *?thesis* **using** *t1-def*(*2*) *mdeg-t21* **by** *simp*
**qed**

**lemma** *num-leaves-1-if-mdeg-1*: *max-deg t* $\leq$ *1* $\Longrightarrow$ *num-leaves t* = *1*
**proof**(*induction t*)
**case** (*Node r xs*)
**then show** *?case*
**proof**(*cases max-deg* (*Node r xs*) = *0*)
**case** *True*
**then show** *?thesis* **using** *empty-iff-mdeg-0* **by** *auto*
**next**
**case** *False*
**then have** *max-deg* (*Node r xs*) = *1* **using** *Node.prems* **by** *simp*
**then obtain** *t e* **where** *t-def*: *xs* = {|(*t*,*e*)|} (*t*,*e*) $\in$ *fset xs*
**using** *mdeg-1-singleton* **by** *fastforce*
**then have** *max-deg t* $\leq$ *1* **using** *Node.prems mdeg-ge-child* **by** *fastforce*
**then show** *?thesis* **using** *Node.IH t-def*(*1*) **by** *simp*
**qed**
**qed**

**lemma** *num-leaves-ge1*: *num-leaves t* $\geq$ *1*
**proof**(*induction t*)
**case** (*Node r xs*)
**show** *?case*
**proof**(*cases xs* = {||})
**case** *False*
**then obtain** *t e* **where** *t-def*: (*t*,*e*) $\in$ *fset xs* **by** *fast*
**then have** *1* $\leq$ *num-leaves t* **using** *Node* **by** *simp*
**then show** *?thesis*
**using** *fset-sum-ge-elem*[*OF finite-fset*[*of xs*] *t-def*, *of* $\lambda$(*t*,*e*). *num-leaves t*] **by** *auto*
**qed** (*simp*)
**qed**

**lemma** *num-leaves-ge-card*: *num-leaves* (*Node r xs*) $\geq$ *fcard xs*
**proof**(*cases xs* = {||})
**case** *False*
**have** *fcard xs* = ($\sum$ *x*$\in$ *fset xs*. *1*) **using** *fcard.rep-eq* **by** *auto*
**also have** ... $\leq$ ($\sum$ *x*$\in$ *fset xs*. *num-leaves* (*fst x*)) **using** *num-leaves-ge1 sum-mono* **by** *metis*
**finally show** *?thesis* **using** *False* **by** (*simp add*: *fst-def prod.case-distrib*)
**qed** (*simp add*: *fcard-fempty*)

**lemma** *num-leaves-root*: *num-leaves* (*Node r xs*) = *num-leaves* (*Node r*′ *xs*)
**by** *simp*

**lemma** *num-leaves-singleton*: *num-leaves* (*Node r* {|(*t*,*e*)|}) = *num-leaves t*
**by** *simp*

## 7.5 List Conversions

**function** *dtree-to-list* :: (*'a*,*'b*) *dtree* ⇒ (*'a*×*'b*) *list* **where**
  *dtree-to-list* (*Node r* {|(*t*,*e*)|}) = (*root t*,*e*) # *dtree-to-list t*
| ∀ *x*. *xs* ≠ {|*x*|} ⟹ *dtree-to-list* (*Node r xs*) = []
  **by** (*metis darcs-mset.cases surj-pair*) *auto*
**termination by** *lexicographic-order*

**fun** *dtree-from-list* :: *'a* ⇒ (*'a*×*'b*) *list* ⇒ (*'a*,*'b*) *dtree* **where**
  *dtree-from-list r* [] = *Node r* {||}
| *dtree-from-list r* ((*v*,*e*)#*xs*) = *Node r* {|(*dtree-from-list v xs*, *e*)|}

**fun** *wf-list-arcs* :: (*'a*×*'b*) *list* ⇒ *bool* **where**
  *wf-list-arcs* [] = *True*
| *wf-list-arcs* ((*v*,*e*)#*xs*) = (*e* ∉ *snd* ‘ *set xs* ∧ *wf-list-arcs xs*)

**fun** *wf-list-verts* :: (*'a*×*'b*) *list* ⇒ *bool* **where**
  *wf-list-verts* [] = *True*
| *wf-list-verts* ((*v*,*e*)#*xs*) = (*v* ∉ *fst* ‘ *set xs* ∧ *wf-list-verts xs*)

**lemma** *dtree-to-list-sub-dverts-ins*:
  *insert* (*root t*) (*fst* ‘ *set* (*dtree-to-list t*)) ⊆ *dverts t*
**proof**(*induction t*)
  **case** (*Node r xs*)
  **show** *?case*
  **proof**(*cases* ∀ *x*. *xs* ≠ {|*x*|})
    **case** *False*
    **then obtain** *t e* **where** *t-def*: *xs* = {|(*t*,*e*)|}
      **using** *mdeg-1-singleton* **by** *fastforce*
    **then show** *?thesis* **using** *Node.IH* **by** *fastforce*
  **qed** (*auto*)
**qed**

**lemma** *dtree-to-list-eq-dverts-ins*:
  *max-deg t* ≤ *1* ⟹ *insert* (*root t*) (*fst* ‘ *set* (*dtree-to-list t*)) = *dverts t*
**proof**(*induction t*)
  **case** (*Node r xs*)
  **show** *?case*
  **proof**(*cases max-deg* (*Node r xs*) = *0*)
    **case** *True*
    **then have** *xs* = {||} **using** *empty-iff-mdeg-0* **by** *auto*
    **moreover from** *this* **have** ∀ *x*. *xs* ≠ {|*x*|} **by** *blast*
    **ultimately show** *?thesis* **by** *simp*
  **next**
    **case** *False*
    **then have** *max-deg* (*Node r xs*) = *1* **using** *Node.prems* **by** *simp*
    **then obtain** *t e* **where** *t-def*: *xs* = {|(*t*,*e*)|} (*t*,*e*) ∈ *fset xs*
      **using** *mdeg-1-singleton* **by** *fastforce*
    **then have** *max-deg t* ≤ *1* **using** *Node.prems mdeg-ge-child* **by** *fastforce*
    **then have** *insert* (*root t*) (*fst* ‘ *set* (*dtree-to-list t*)) = *dverts t*

```
    using Node.IH t-def(2) by auto
  then show ?thesis using Node.prems(1) t-def(1) by simp
 qed
qed

lemma dtree-to-list-eq-dverts-sucs:
 max-deg t ≤ 1 ⟹ fst ‘ set (dtree-to-list t) = (⋃ x ∈ fset (sucs t). dverts (fst x))
proof(induction t)
 case (Node r xs)
 show ?case
 proof(cases max-deg (Node r xs) = 0)
  case True
  then have xs = {||} using empty-iff-mdeg-0 by auto
  moreover from this have ∀ x. xs ≠ {|x|} by blast
  ultimately show ?thesis by simp
 next
  case False
  then have max-deg (Node r xs) = 1 using Node.prems by simp
  then obtain t e where t-def: xs = {|(t,e)|} (t,e) ∈ fset xs
   using mdeg-1-singleton by fastforce
  then have max-deg t ≤ 1 using Node.prems mdeg-ge-child by fastforce
  then have fst ‘ set (dtree-to-list t) = (⋃ x ∈ fset (sucs t). dverts (fst x))
   using Node.IH t-def(2) by auto
  moreover from this have dverts t = insert (root t) (⋃ x ∈ fset (sucs t). dverts
(fst x))
   using ‹max-deg t ≤ 1› dtree-to-list-eq-dverts-ins by fastforce
  ultimately show ?thesis using Node.prems(1) t-def(1) by force
 qed
qed

lemma dtree-to-list-sub-dverts:
 wf-dverts t ⟹ fst ‘ set (dtree-to-list t) ⊆ dverts t − {root t}
proof(induction t)
 case (Node r xs)
 show ?case
 proof(cases ∀ x. xs ≠ {|x|})
  case False
  then obtain t e where t-def: xs = {|(t,e)|}
   using mdeg-1-singleton by fastforce
  then have wf-dverts t using Node.prems mdeg-ge-child by fastforce
  then have fst ‘ set (dtree-to-list t) ⊆ dverts t − {root t} using Node.IH t-def(1)
by auto
  then have fst ‘ set (dtree-to-list (Node r xs)) ⊆ dverts t
   using t-def(1) dtree.set-sel(1) by auto
  then show ?thesis using Node.prems(1) t-def(1) by (simp add: wf-dverts-iff-dverts′)
 qed (auto)
qed

lemma dtree-to-list-eq-dverts:
```

```
  ⟦wf-dverts t; max-deg t ≤ 1⟧ ⟹ fst ‘ set (dtree-to-list t) = dverts t − {root t}
proof(induction t)
  case (Node r xs)
  show ?case
  proof(cases max-deg (Node r xs) = 0)
    case True
    then have xs = {||} using empty-iff-mdeg-0 by auto
    moreover from this have ∀ x. xs ≠ {|x|} by blast
    ultimately show ?thesis by simp
  next
    case False
    then have max-deg (Node r xs) = 1 using Node.prems by simp
    then obtain t e where t-def: xs = {|(t,e)|} (t,e) ∈ fset xs
      using mdeg-1-singleton by fastforce
    then have max-deg t ≤ 1 ∧ wf-dverts t using Node.prems mdeg-ge-child by
fastforce
   then have fst ‘ set (dtree-to-list t) = dverts t − {root t} using Node.IH t-def(2)
by auto
    then have fst ‘ set (dtree-to-list (Node r xs)) = dverts t
      using t-def(1) dtree.set-sel(1) by auto
   then show ?thesis using Node.prems(1) t-def(1) by (simp add: wf-dverts-iff-dverts′)
  qed
qed

lemma dtree-to-list-eq-dverts-single:
  ⟦max-deg t ≤ 1; sucs t = {|(t1,e1)|}⟧ ⟹ fst ‘ set (dtree-to-list t) = dverts t1
  by (simp add: dtree-to-list-eq-dverts-sucs)

lemma dtree-to-list-sub-darcs: snd ‘ set (dtree-to-list t) ⊆ darcs t
proof(induction t)
  case (Node r xs)
  show ?case
  proof(cases ∀ x. xs ≠ {|x|})
    case False
    then obtain t e where xs = {|(t,e)|}
      using mdeg-1-singleton by fastforce
    then show ?thesis using Node.IH by fastforce
  qed (auto)
qed

lemma dtree-to-list-eq-darcs: max-deg t ≤ 1 ⟹ snd ‘ set (dtree-to-list t) = darcs
t
proof(induction t)
  case (Node r xs)
  show ?case
  proof(cases max-deg (Node r xs) = 0)
    case True
    then have xs = {||} using empty-iff-mdeg-0 by auto
    moreover from this have ∀ x. xs ≠ {|x|} by blast
```

```
    ultimately show ?thesis by simp
  next
    case False
    then have max-deg (Node r xs) = 1 using Node.prems by simp
    then obtain t e where t-def: xs = {|(t,e)|} (t,e) ∈ fset xs
      using mdeg-1-singleton by fastforce
    then have max-deg t ≤ 1 using Node.prems mdeg-ge-child by fastforce
    then have snd ‘ set (dtree-to-list t) = darcs t using Node.IH t-def(2) by auto
    then show ?thesis using t-def(1) by simp
  qed
qed

lemma dtree-from-list-eq-dverts: dverts (dtree-from-list r xs) = insert r (fst ‘ set
xs)
  by(induction xs arbitrary: r) force+

lemma dtree-from-list-eq-darcs: darcs (dtree-from-list r xs) = snd ‘ set xs
  by(induction xs arbitrary: r) force+

lemma dtree-from-list-root-r[simp]: root (dtree-from-list r xs) = r
  using dtree.sel(1) dtree-from-list.elims by metis

lemma dtree-from-list-v-eq-r:
  Node r xs = dtree-from-list v ys ⟹ r = v
  using dtree.sel(1)[of r xs] by simp

lemma dtree-from-list-fcard0-empty: fcard (sucs (dtree-from-list r [])) = 0
  by simp

lemma dtree-from-list-fcard0-iff-empty: fcard (sucs (dtree-from-list r xs)) = 0 ⟷
xs = []
  by(induction xs) auto

lemma dtree-from-list-fcard1-iff-nempty: fcard (sucs (dtree-from-list r xs)) = 1
⟷ xs ≠ []
  by(induction xs) (auto simp: fcard-single-1 fcard-fempty)

lemma dtree-from-list-fcard-le1: fcard (sucs (dtree-from-list r xs)) ≤ 1
  by(induction xs) (auto simp: fcard-single-1 fcard-fempty)

lemma dtree-from-empty-deg-0: max-deg (dtree-from-list r []) = 0
  by simp

lemma dtree-from-list-deg-le-1: max-deg (dtree-from-list r xs) ≤ 1
proof(induction xs arbitrary: r)
  case Nil
  have max-deg (dtree-from-list r []) = 0 by simp
  also have ... ≤ 1 by blast
  finally show ?case by blast
```

**next**
**case** (*Cons x xs*)
**obtain** *v e* **where** *v-def*: *x* = (*v*,*e*) **by** *force*
**let** *?xs* = {|(*dtree-from-list v xs*, *e*)|}
**have** *dtree-from-list r* (*x*#*xs*) = *Node r ?xs* **by** (*simp add*: *v-def*)
**moreover have** *max-deg* (*dtree-from-list v xs*) ≤ *1* **using** *Cons* **by** *simp*
**moreover have** *max-deg* (*Node r ?xs*) = *max* (*max-deg* (*dtree-from-list v xs*)) (*fcard ?xs*)
**using** *mdeg-singleton* **by** *fast*
**ultimately show** *?case* **by** (*simp add*: *fcard-finsert-if max-def*)
**qed**

**lemma** *dtree-from-list-deg-1*: *xs* ≠ [] ⟷ *max-deg* (*dtree-from-list r xs*) = *1*
**proof** (*cases xs*)
**case** (*Cons x xs*)
**obtain** *v e* **where** *v-def*: *x* = (*v*,*e*) **by** *force*
**let** *?xs* = {|(*dtree-from-list v xs*, *e*)|}
**have** *dtree-from-list r* (*x*#*xs*) = *Node r ?xs* **by** (*simp add*: *v-def*)
**moreover have** *max-deg* (*dtree-from-list v xs*) ≤ *1* **using** *dtree-from-list-deg-le-1* **by** *fast*
**moreover have** *max-deg* (*Node r ?xs*) = *max* (*max-deg* (*dtree-from-list v xs*)) (*fcard ?xs*)
**using** *mdeg-singleton* **by** *fast*
**ultimately show** *?thesis* **using** *Cons* **by** (*simp add*: *fcard-finsert-if max-def*)
**qed** (*metis dtree-from-empty-deg-0 zero-neq-one*)

**lemma** *dtree-from-list-singleton*: *xs* ≠ [] ⟹ ∃ *t e*. *dtree-from-list r xs* = *Node r* {|(*t*,*e*)|}
**using** *dtree-from-list.elims*[*of r xs*] **by** *fastforce*

**lemma** *dtree-from-to-list-id*: *max-deg t* ≤ *1* ⟹ *dtree-from-list* (*root t*) (*dtree-to-list t*) = *t*
**proof**(*induction t*)
**case** (*Node r xs*)
**then show** *?case*
**proof**(*cases max-deg* (*Node r xs*) = *0*)
**case** *True*
**then have** *xs* = {||} **using** *empty-iff-mdeg-0* **by** *auto*
**moreover from** *this* **have** ∀ *x*. *xs* ≠ {|*x*|} **by** *blast*
**ultimately show** *?thesis* **using** *Node.prems* **by** *simp*
**next**
**case** *False*
**then have** *max-deg* (*Node r xs*) = *1* **using** *Node.prems* **by** *simp*
**then obtain** *t e* **where** *t-def*: *xs* = {|(*t*,*e*)|} (*t*,*e*) ∈ *fset xs*
**using** *mdeg-1-singleton* **by** *fastforce*
**then have** *max-deg t* ≤ *1* **using** *Node.prems mdeg-ge-child* **by** *fastforce*
**then show** *?thesis* **using** *Node.IH t-def*(*1*) **by** *simp*
**qed**
**qed**

**lemma** *dtree-to-from-list-id*: *dtree-to-list* (*dtree-from-list r xs*) = *xs*
**proof**(*induction xs arbitrary*: *r*)
**case** *Nil*
**then show** *?case*
**using** *dtree-from-list-deg-1 dtree-from-list-deg-le-1 dtree-from-to-list-id* **by** *metis*
**next**
**case** (*Cons x xs*)
**obtain** *v e* **where** *v-def*: *x* = (*v*,*e*) **by** *force*
**then have** *dtree-to-list* (*dtree-from-list r* (*x*#*xs*)) = (*v*,*e*)#*dtree-to-list* (*dtree-from-list v xs*)
**by** (*metis dtree-from-list.elims dtree-to-list.simps*(*1*) *dtree.sel*(*1*) *dtree-from-list.simps*(*2*))
**then show** *?case* **by** (*simp add*: *v-def Cons*)
**qed**

**lemma** *dtree-from-list-eq-singleton-hd*:
*Node r0* {|(*t0*,*e0*)|} = *dtree-from-list v1 ys* ⟹ (∃ *xs*. (*root t0*, *e0*) # *xs* = *ys*)
**using** *dtree-to-list.simps*(*1*)[*of r0 t0 e0*] *dtree-to-from-list-id*[*of v1 ys*] **by** *simp*

**lemma** *dtree-from-list-eq-singleton*:
*Node r0* {|(*t0*,*e0*)|} = *dtree-from-list v1 ys* ⟹ *r0* = *v1* ∧ (∃ *xs*. (*root t0*, *e0*) # *xs* = *ys*)
**using** *dtree-from-list-eq-singleton-hd* **by** *fastforce*

**lemma** *dtree-from-list-uneq-sequence*:
⟦*is-subtree* (*Node r0* {|(*t0*,*e0*)|}) (*dtree-from-list v1 ys*);
*Node r0* {|(*t0*,*e0*)|} ≠ *dtree-from-list v1 ys*⟧
⟹ ∃ *e as bs*. *as* @ (*r0*,*e*) # (*root t0*, *e0*) # *bs* = *ys*
**proof**(*induction v1 ys rule*: *dtree-from-list.induct*)
**case** (*2 r v e xs*)
**then show** *?case*
**proof**(*cases Node r0* {|(*t0*,*e0*)|} = *dtree-from-list v xs*)
**case** *True*
**then show** *?thesis* **using** *dtree-from-list-eq-singleton* **by** *fast*
**next**
**case** *False*
**then obtain** *e1 as bs* **where** *as* @ (*r0*, *e1*) # (*root t0*, *e0*) # *bs* = *xs* **using** *2* **by** *auto*
**then have** ((*v*,*e*)#*as*) @ (*r0*, *e1*) # (*root t0*, *e0*) # *bs* = (*v*, *e*) # *xs* **by** *simp*
**then show** *?thesis* **by** *blast*
**qed**
**qed**(*simp*)

**lemma** *dtree-from-list-sequence*:
⟦*is-subtree* (*Node r0* {|(*t0*,*e0*)|}) (*dtree-from-list v1 ys*)⟧
⟹ ∃ *e as bs*. *as* @ (*r0*,*e*) # (*root t0*, *e0*) # *bs* = ((*v1*,*e1*)#*ys*)
**using** *dtree-from-list-uneq-sequence*[*of r0 t0 e0*] *dtree-from-list-eq-singleton append-Cons* **by** *fast*

**lemma** *dtree-from-list-eq-empty*:
*Node r* {||} = *dtree-from-list v ys* ⟹ *r* = *v* ∧ *ys* = []
**using** *dtree-to-from-list-id dtree-from-list-v-eq-r dtree-from-list.simps*(*1*) **by** *metis*

**lemma** *dtree-from-list-sucs-cases*:
*Node r xs* = *dtree-from-list v ys* ⟹ *xs* = {||} ∨ (∃ *x*. *xs* = {|*x*|})
**using** *dtree.inject dtree-from-list.simps*(*1*) *dtree-to-from-list-id dtree-to-list.simps*(*2*)
**by** *metis*

**lemma** *dtree-from-list-uneq-sequence-xs*:
*strict-subtree* (*Node r0 xs0*) (*dtree-from-list v1 ys*)
⟹ ∃ *e as bs*. *as* @ (*r0*,*e*) # *bs* = *ys* ∧ *Node r0 xs0* = *dtree-from-list r0 bs*
**proof**(*induction v1 ys rule*: *dtree-from-list.induct*)
**case** (*2 r v e xs*)
**then show** *?case*
**proof**(*cases Node r0 xs0* = *dtree-from-list v xs*)
**case** *True*
**then show** *?thesis* **using** *dtree-from-list-root-r dtree.sel*(*1*)[*of r0 xs0*] **by** *fast-force*
**next**
**case** *False*
**then obtain** *e1 as bs* **where** *0*: *as* @ (*r0*,*e1*) # *bs* = *xs Node r0 xs0* = *dtree-from-list r0 bs*
**using** *2* **unfolding** *strict-subtree-def* **by** *auto*
**then have** ((*v*,*e*)#*as*) @ (*r0*,*e1*) # *bs* = (*v*,*e*) # *xs* **by** *simp*
**then show** *?thesis* **using** *0*(*2*) **by** *blast*
**qed**
**qed**(*simp add*: *strict-subtree-def*)

**lemma** *dtree-from-list-sequence-xs*:
⟦*is-subtree* (*Node r xs*) (*dtree-from-list v1 ys*)⟧
⟹ ∃ *e as bs*. *as* @ (*r*,*e*) # *bs* = ((*v1*,*e1*)#*ys*) ∧ *Node r xs* = *dtree-from-list r bs*
**using** *dtree-from-list-uneq-sequence-xs*[*of r xs*] *dtree-from-list-v-eq-r strict-subtree-def*
**by** (*fast intro*!: *append-Cons*)

**lemma** *dtree-from-list-sequence-dverts*:
⟦*is-subtree* (*Node r xs*) (*dtree-from-list v1 ys*)⟧
⟹ ∃ *e as bs*. *as* @ (*r*,*e*) # *bs* = ((*v1*,*e1*)#*ys*) ∧ *dverts* (*Node r xs*) = *insert r* (*fst* ‘ *set bs*)
**using** *dtree-from-list-sequence-xs*[*of r xs v1 ys e1*] *dtree-from-list-eq-dverts* **by** *metis*

**lemma** *dtree-from-list-dverts-subset-set*:
*set bs* ⊆ *set ds* ⟹ *dverts* (*dtree-from-list r bs*) ⊆ *dverts* (*dtree-from-list r ds*)
**by** (*auto simp*: *dtree-from-list-eq-dverts*)

**lemma** *wf-darcs'-iff-wf-list-arcs*: *wf-list-arcs xs* ⟷ *wf-darcs'* (*dtree-from-list r xs*)
**by**(*induction xs arbitrary*: *r rule*: *wf-list-arcs.induct*) (*auto simp*: *dtree-from-list-eq-darcs*)

**lemma** *wf-darcs-iff-wf-list-arcs*: *wf-list-arcs xs* $\longleftrightarrow$ *wf-darcs* (*dtree-from-list r xs*)
**using** *wf-darcs'-iff-wf-list-arcs wf-darcs-iff-darcs'* **by** *fast*

**lemma** *wf-dverts-iff-wf-list-verts*:
$r \notin$ *fst* ' *set xs* $\wedge$ *wf-list-verts xs* $\longleftrightarrow$ *wf-dverts* (*dtree-from-list r xs*)
**by** (*induction xs arbitrary*: *r rule*: *wf-list-verts.induct*)
(*auto simp*: *dtree-from-list-eq-dverts wf-dverts-iff-dverts'*)

**theorem** *wf-dtree-iff-wf-list*:
*wf-list-arcs xs* $\wedge$ $r \notin$ *fst* ' *set xs* $\wedge$ *wf-list-verts xs* $\longleftrightarrow$ *wf-dtree* (*dtree-from-list r xs*)
**using** *wf-darcs-iff-wf-list-arcs wf-dverts-iff-wf-list-verts* **unfolding** *wf-dtree-def* **by** *fast*

**lemma** *wf-list-arcs-if-wf-darcs*: *wf-darcs t* $\Longrightarrow$ *wf-list-arcs* (*dtree-to-list t*)
**proof**(*induction t*)
**case** (*Node r xs*)
**then show** *?case*
**proof**(*cases* $\forall x.$ *xs* $\neq$ {|*x*|})
**case** *True*
**then show** *?thesis* **using** *dtree-to-list.simps*(*2*) **by** *simp*
**next**
**case** *False*
**then obtain** *t1 e1* **where** *xs* = {|(*t1*,*e1*)|} **by** *auto*
**then show** *?thesis*
**using** *Node dtree-to-list-sub-darcs* **unfolding** *wf-darcs-iff-darcs'* **by** *fastforce*
**qed**
**qed**

**lemma** *wf-list-verts-if-wf-dverts*: *wf-dverts t* $\Longrightarrow$ *wf-list-verts* (*dtree-to-list t*)
**proof**(*induction t*)
**case** (*Node r xs*)
**then show** *?case*
**proof**(*cases* $\forall x.$ *xs* $\neq$ {|*x*|})
**case** *True*
**then show** *?thesis* **using** *dtree-to-list.simps*(*2*) **by** *simp*
**next**
**case** *False*
**then obtain** *t1 e1* **where** *xs* = {|(*t1*,*e1*)|} **by** *auto*
**then show** *?thesis* **using** *Node dtree-to-list-sub-dverts* **by** (*fastforce simp*: *wf-dverts-iff-dverts'*)
**qed**
**qed**

**lemma** *distinct-if-wf-list-arcs*: *wf-list-arcs xs* $\Longrightarrow$ *distinct xs*
**by** (*induction xs*) *force*+

**lemma** *distinct-if-wf-list-verts*: *wf-list-verts xs* $\Longrightarrow$ *distinct xs*

**by** (*induction xs*) *force+*

**lemma** *wf-list-arcs-alt*: *wf-list-arcs xs* $\longleftrightarrow$ *distinct* (*map snd xs*)
**by** (*induction xs*) *force+*

**lemma** *wf-list-verts-alt*: *wf-list-verts xs* $\longleftrightarrow$ *distinct* (*map fst xs*)
**by** (*induction xs*) *force+*

**lemma** *subtree-from-list-split-eq-if-wfverts*:
**assumes** *wf-list-verts* (*as*@(*r*,*e*)#*bs*)
**and** *v* $\notin$ *fst* ' *set* (*as*@(*r*,*e*)#*bs*)
**and** *is-subtree* (*Node r xs*) (*dtree-from-list v* (*as*@(*r*,*e*)#*bs*))
**shows** *Node r xs* = *dtree-from-list r bs*
**proof** −
**have** *0*: *wf-list-verts* ((*v*,*e*)#*as*@(*r*,*e*)#*bs*) **using** *assms*(*1*,*2*) **by** *simp*
**obtain** *as*′ *e*′ *bs*′ **where** *as*′*-def*:
*as*′@(*r*,*e*′)#*bs*′ = (*v*,*e*)#*as*@(*r*,*e*)#*bs Node r xs* = *dtree-from-list r bs*′
**using** *assms*(*3*) *dtree-from-list-sequence-xs*[*of r xs*] **by** *blast*
**then have** *0*: *wf-list-verts* (*as*′@(*r*,*e*′)#*bs*′) **using** *assms*(*1*,*2*) **by** *simp*
**have** *r-as*′: *r* $\notin$ *fst* ' *set as*′ **using** *0* **unfolding** *wf-list-verts-alt* **by** *simp*
**moreover have** *r-bs*′: *r* $\notin$ *fst* ' *set bs*′ **using** *0* **unfolding** *wf-list-verts-alt* **by** *simp*
**moreover have** (*r*,*e*) ∈ *set* (*as*′@(*r*,*e*′)#*bs*′) **using** *as*′*-def*(*1*) **by** *simp*
**ultimately have** (*r*,*e*′)= (*r*,*e*) **by** *force*
**then show** *?thesis*
**using** *r-as*′ *r-bs*′ *as*′*-def append-Cons-eq-iff*[*of* (*r*,*e*) *as*′ *bs*′ (*v*,*e*)#*as bs*] **by** *force*
**qed**

**lemma** *subtree-from-list-split-eq-if-wfdverts*:
⟦*wf-dverts* (*dtree-from-list v* (*as*@(*r*,*e*)#*bs*));
*is-subtree* (*Node r xs*) (*dtree-from-list v* (*as*@(*r*,*e*)#*bs*))⟧
$\Longrightarrow$ *Node r xs* = *dtree-from-list r bs*
**using** *subtree-from-list-split-eq-if-wfverts wf-dverts-iff-wf-list-verts* **by** *fast*

**lemma** *dtree-from-list-dverts-subset-wfdverts*:
**assumes** *set bs* ⊆ *set ds*
**and** *wf-dverts* (*dtree-from-list v* (*as*@(*r*,*e1*)#*bs*))
**and** *wf-dverts* (*dtree-from-list v* (*cs*@(*r*,*e2*)#*ds*))
**and** *is-subtree* (*Node r xs*) (*dtree-from-list v* (*as*@(*r*,*e1*)#*bs*))
**and** *is-subtree* (*Node r ys*) (*dtree-from-list v* (*cs*@(*r*,*e2*)#*ds*))
**shows** *dverts* (*Node r xs*) ⊆ *dverts* (*Node r ys*)
**using** *dtree-from-list-dverts-subset-set*[*OF assms*(*1*)]
*subtree-from-list-split-eq-if-wfdverts*[*OF assms*(*2*,*4*)]
*subtree-from-list-split-eq-if-wfdverts*[*OF assms*(*3*,*5*)]
**by** *simp*

**lemma** *dtree-from-list-dverts-subset-wfdverts*′:
**assumes** *wf-dverts* (*dtree-from-list v as*)

**and** *wf-dverts* (*dtree-from-list v cs*)
**and** *is-subtree* (*Node r xs*) (*dtree-from-list v as*)
**and** *is-subtree* (*Node r ys*) (*dtree-from-list v cs*)
**and** ∃ *as′ e1 bs cs′ e2 ds. as′*@(*r*,*e1*)#*bs* = *as* ∧ *cs′*@(*r*,*e2*)#*ds* = *cs* ∧ *set bs* ⊆ *set ds*
**shows** *dverts* (*Node r xs*) ⊆ *dverts* (*Node r ys*)
**using** *dtree-from-list-dverts-subset-wfdverts assms* **by** *metis*

**lemma** *dtree-to-list-sequence-subtree*:
⟦*max-deg t* ≤ *1*; *strict-subtree* (*Node r xs*) *t*⟧
⟹ ∃ *as e bs. dtree-to-list t* = *as*@(*r*,*e*)#*bs* ∧ *Node r xs* = *dtree-from-list r bs*
**by** (*metis dtree-from-list-uneq-sequence-xs dtree-from-to-list-id*)

**lemma** *dtree-to-list-sequence-subtree′*:
⟦*max-deg t* ≤ *1*; *strict-subtree* (*Node r xs*) *t*⟧
⟹ ∃ *as e bs. dtree-to-list t* = *as*@(*r*,*e*)#*bs* ∧ *dtree-to-list* (*Node r xs*) = *bs*
**using** *dtree-to-from-list-id*[*of r*] *dtree-to-list-sequence-subtree*[*of t r xs*] **by** *fastforce*

**lemma** *dtree-to-list-subtree-dverts-eq-fsts*:
⟦*max-deg t* ≤ *1*; *strict-subtree* (*Node r xs*) *t*⟧
⟹ ∃ *as e bs. dtree-to-list t* = *as*@(*r*,*e*)#*bs* ∧ *insert r* (*fst* ‘ *set bs*) = *dverts* (*Node r xs*)
**by** (*metis dtree-from-list-eq-dverts dtree-to-list-sequence-subtree*)

**lemma** *dtree-to-list-subtree-dverts-eq-fsts′*:
⟦*max-deg t* ≤ *1*; *strict-subtree* (*Node r xs*) *t*⟧
⟹ ∃ *as e bs. dtree-to-list t* = *as*@(*r*,*e*)#*bs* ∧ (*fst* ‘ *set* ((*r*,*e*)#*bs*)) = *dverts* (*Node r xs*)
**using** *dtree-to-list-subtree-dverts-eq-fsts* **by** *fastforce*

**lemma** *dtree-to-list-split-subtree*:
**assumes** *as*@(*r*,*e*)#*bs* = *dtree-to-list t*
**shows** ∃ *xs. strict-subtree* (*Node r xs*) *t* ∧ *dtree-to-list* (*Node r xs*) = *bs*
**using** *assms* **proof**(*induction t arbitrary*: *as rule*: *dtree-to-list.induct*)
**case** (*1 r1 t1 e1*)
**show** *?case*
**proof**(*cases as*)
**case** *Nil*
**then have** *dtree-to-list* (*Node r* (*sucs t1*)) = *bs* **using** *1.prems* **by** *auto*
**moreover have** *is-subtree* (*Node r* (*sucs t1*)) (*Node r1* {|(*t1*, *e1*)|})
**using** *subtree-if-child*[*of t1* {|(*t1*, *e1*)|}] *1.prems Nil* **by** *simp*
**moreover have** *Node r1* {|(*t1*, *e1*)|} ≠ (*Node r* (*sucs t1*)) **by** (*blast intro*!: *singleton-uneq′*)
**ultimately show** *?thesis* **unfolding** *strict-subtree-def* **by** *blast*
**next**
**case** (*Cons a as′*)
**then show** *?thesis* **using** *1* **unfolding** *strict-subtree-def* **by** *fastforce*
**qed**

**qed**(*simp*)

**lemma** *dtree-to-list-split-subtree-dverts-eq-fsts*:
**assumes** *max-deg t* ≤ *1* **and** *as*@(*r*,*e*)#*bs* = *dtree-to-list t*
**shows** ∃ *xs*. *strict-subtree* (*Node r xs*) *t* ∧ *dverts* (*Node r xs*) = *insert r* (*fst*‘*set bs*)
**proof** −
**obtain** *xs* **where** *xs-def*:
*is-subtree* (*Node r xs*) *t Node r xs* ≠ *t dtree-to-list* (*Node r xs*) = *bs*
**using** *dtree-to-list-split-subtree*[*OF assms*(*2*)] **unfolding** *strict-subtree-def* **by** *blast*
**have** *max-deg* (*Node r xs*) ≤ *1* **using** *mdeg-ge-sub*[*OF xs-def*(*1*)] *assms*(*1*) **by** *simp*
**then show** *?thesis*
**using** *dtree-to-list-eq-dverts-ins*[*of Node r xs*] *xs-def strict-subtree-def* **by** *auto*
**qed**

**lemma** *dtree-to-list-split-subtree-dverts-eq-fsts′*:
**assumes** *max-deg t* ≤ *1* **and** *as*@(*r*,*e*)#*bs* = *dtree-to-list t*
**shows** ∃ *xs*. *strict-subtree* (*Node r xs*) *t* ∧ *dverts* (*Node r xs*) = (*fst* ‘ *set* ((*r*,*e*)#*bs*))
**using** *dtree-to-list-split-subtree-dverts-eq-fsts*[*OF assms*] **by** *simp*

**lemma** *dtree-from-list-dverts-subset-wfdverts1*:
**assumes** *dverts t1* ⊆ *fst* ‘ *set* ((*r*,*e2*)#*bs*)
**and** *wf-dverts* (*dtree-from-list v* (*as*@(*r*,*e2*)#*bs*))
**and** *is-subtree* (*Node r ys*) (*dtree-from-list v* (*as*@(*r*,*e2*)#*bs*))
**shows** *dverts t1* ⊆ *dverts* (*Node r ys*)
**using** *subtree-from-list-split-eq-if-wfdverts*[*OF assms*(*2*,*3*)] *assms*(*1*) *dtree-from-list-eq-dverts*
**by** *fastforce*

**lemma** *dtree-from-list-dverts-subset-wfdverts1′*:
**assumes** *wf-dverts* (*dtree-from-list v cs*)
**and** *is-subtree* (*Node r ys*) (*dtree-from-list v cs*)
**and** ∃ *as e bs*. *as*@(*r*,*e*)#*bs* = *cs* ∧ *dverts t1* ⊆ *fst* ‘ *set* ((*r*,*e*)#*bs*)
**shows** *dverts t1* ⊆ *dverts* (*Node r ys*)
**using** *dtree-from-list-dverts-subset-wfdverts1 assms* **by** *fast*

**lemma** *dtree-from-list-1-leaf*: *num-leaves* (*dtree-from-list r xs*) = *1*
**using** *num-leaves-1-if-mdeg-1 dtree-from-list-deg-le-1* **by** *fast*

## 7.6 Inserting in Dtrees

**abbreviation** *insert-before* ::
*′a* ⇒ *′b* ⇒ *′a* ⇒ ((*′a*,*′b*) *dtree* × *′b*) *fset* ⇒ ((*′a*,*′b*) *dtree* × *′b*) *fset* **where**
*insert-before v e y xs* ≡ *ffold* (λ(*t1*,*e1*).
*finsert* (*if root t1* = *y then* (*Node v* {|(*t1*,*e1*)|},*e*) *else* (*t1*,*e1*))) {||} *xs*

**fun** *insert-between* :: *′a* ⇒ *′b* ⇒ *′a* ⇒ *′a* ⇒ (*′a*,*′b*) *dtree* ⇒ (*′a*,*′b*) *dtree* **where**
*insert-between v e x y* (*Node r xs*) = (*if x*=*r* ∧ (∃ *t*. *t* ∈ *fst* ‘ *fset xs* ∧ *root t* = *y*)

*then Node r (insert-before v e y xs)*
*else if x=r then Node r (finsert (Node v {||},e) xs)*
*else Node r ((λ(t,e1). (insert-between v e x y t,e1)) |‘| xs))*

**lemma** *insert-between-id-if-notin*: *x ∉ dverts t ⟹ insert-between v e x y t = t*
**proof**(*induction t*)
**case** (*Node r xs*)
**have** *∀(t,e) ∈ fset xs. x ∉ dverts t* **using** *Node.prems* **by** *force*
**then have** *∀(t,e1) ∈ fset xs. (λ(t,e1). (insert-between v e x y t,e1)) (t,e1) = (t,e1)*
**using** *Node.IH* **by** *auto*
**then have** *((λ(t,e1). (insert-between v e x y t,e1)) |‘| xs) = xs*
**by** (*smt* (*verit*, *ccfv-threshold*) *fset.map-cong0 case-prodE fimage-ident*)
**then show** *?case* **using** *Node.prems* **by** *simp*
**qed**

**context** *wf-dtree*
**begin**

**lemma** *insert-before-commute-aux*:
**assumes** *f = (λ(t1,e1). finsert (if root t1 = y1 then (Node v {|(t1,e1)|},e) else (t1,e1)))*
**shows** *(f y ∘ f x) z = (f x ∘ f y) z*
**proof** −
**obtain** *t1 e1* **where** *y-def*: *y = (t1, e1)* **by** *fastforce*
**obtain** *t2 e2* **where** *x = (t2, e2)* **by** *fastforce*
**then show** *?thesis* **using** *assms y-def* **by** *auto*
**qed**

**lemma** *insert-before-commute*:
*comp-fun-commute (λ(t1,e1). finsert (if root t1 = y1 then (Node v {|(t1,e1)|},e) else (t1,e1)))*
**using** *comp-fun-commute-def insert-before-commute-aux* **by** *fastforce*

**interpretation** *Comm*:
*comp-fun-commute λ(t1,e1). finsert (if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1))*
**by** (*rule insert-before-commute*)

**lemma** *root-not-new-in-orig*:
*⟦(t1,e1) ∈ fset (insert-before v e y xs); root t1 ≠ v⟧ ⟹ (t1,e1) ∈ fset xs*
**proof**(*induction xs*)
**case** *empty*
**then show** *?case* **by** *simp*
**next**
**case** (*insert x xs*)
**let** *?f = (λ(t1,e1). if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1))*
**show** *?case*
**proof**(*cases (t1,e1) ∈ fset (insert-before v e y xs)*)

```
    case True
    then show ?thesis using insert.IH insert.prems(2) by simp
  next
    case False
    have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
      by (simp add: insert.hyps prod.case-distrib)
    then have ?f x = (t1,e1) using False insert.prems(1) by force
    then have x = (t1,e1)
    by (smt (z3) insert.prems(2) dtree.sel(1) old.prod.exhaust prod.inject case-prod-conv)
    then show ?thesis by simp
  qed
qed

lemma root-not-y-in-new:
  ⟦(t1,e1) ∈ fset xs; root t1 ≠ y⟧ ⟹ (t1,e1) ∈ fset (insert-before v e y xs)
proof(induction xs)
  case empty
  then show ?case by simp
next
  case (insert x xs)
  let ?f = (λ(t1,e1). if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1))
  show ?case
  proof(cases (t1,e1) = x)
    case True
    then show ?thesis using insert by auto
  next
    case False
    have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
      by (simp add: insert.hyps prod.case-distrib)
    then show ?thesis using insert.IH insert.prems by force
  qed
qed

lemma root-noty-if-in-insert-before:
  ⟦(t1,e1) ∈ fset (insert-before v e y xs); v≠y⟧ ⟹ root t1 ≠ y
proof(induction xs)
  case empty
  then show ?case by simp
next
  case (insert x xs)
  let ?f = (λ(t1,e1). if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1))
  show ?case
  proof(cases (t1,e1) ∈ fset (insert-before v e y xs))
    case True
    then show ?thesis using insert.IH insert.prems(2) by fast
  next
    case False
    have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
      by (simp add: insert.hyps prod.case-distrib)
```

```
    then have 0: ?f x = (t1,e1) using insert.prems False by simp
    then show ?thesis
    proof(cases root t1 = v)
      case True
      then show ?thesis using insert.prems(2) by simp
    next
      case False
       then show ?thesis by (smt (z3) dtree.sel(1) old.prod.exhaust prod.inject 0
case-prod-conv)
    qed
  qed
qed

lemma in-insert-before-child-in-orig:
  ⟦(t1,e1) ∈ fset (insert-before v e y xs); (t1,e1) ∉ fset xs⟧
    ⟹ ∃(t2,e2) ∈ fset xs. (Node v {|(t2,e2)|}) = t1 ∧ root t2 = y ∧ e1=e
proof(induction xs)
  case empty
  then show ?case by simp
next
  case (insert x xs)
  let ?f = (λ(t1,e1). if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1))
  show ?case
  proof(cases (t1,e1) ∈ fset (insert-before v e y xs))
    case True
    then show ?thesis using insert.IH insert.prems(2) by simp
  next
    case False
    have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
      by (simp add: insert.hyps prod.case-distrib)
    then show ?thesis
     by (smt (z3) False Pair-inject old.prod.case case-prodI2 finsert-iff insert.prems)
  qed
qed

lemma insert-before-not-y-id:
  ¬(∃ t. t ∈ fst ‘ fset xs ∧ root t = y) ⟹ insert-before v e y xs = xs
proof(induction xs)
  case (insert x xs)
  let ?f = (λ(t1,e1). if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1))
  have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
    by (simp add: insert.hyps prod.case-distrib)
  then have insert-before v e y (finsert x xs) = finsert x (insert-before v e y xs)
    using insert.prems
    by (smt (z3) old.prod.exhaust case-prod-conv finsertCI fst-conv image-eqI)
  moreover have ¬(∃ t. t ∈ fst ‘ fset xs ∧ root t = y) using insert.prems by auto
  ultimately show ?case using insert.IH by blast
qed (simp)
```

```
lemma insert-before-alt:
  insert-before v e y xs
  = (λ(t1,e1). if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1)) |‘| xs
  by(induction xs) (auto simp: Product-Type.prod.case-distrib)

lemma dverts-insert-before-aux:
  ∃ t. t ∈ fst ‘ fset xs ∧ root t = y
    ⟹ (⋃ x∈fset (insert-before v e y xs). ⋃ (dverts ‘ Basic-BNFs.fsts x))
       = insert v (⋃ x∈fset xs. ⋃ (dverts ‘ Basic-BNFs.fsts x))
proof(induction xs)
  case empty
  then show ?case by simp
next
  case (insert x xs)
  let ?f = (λ(t1,e1). if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1))
  obtain t1 e1 where t1-def: x = (t1,e1) by fastforce
  then show ?case
  proof(cases root t1 = y)
    case True
    then have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y
xs)
      by (simp add: insert.hyps prod.case-distrib)
    then have insert-before v e y (finsert x xs)
              = finsert (Node v {|(t1,e1)|},e) (insert-before v e y xs)
      using t1-def True by simp
     then have 0: (⋃ x∈fset (insert-before v e y (finsert x xs)). ⋃ (dverts ‘ Ba-
sic-BNFs.fsts x))
        = insert v (dverts t1) ∪ (⋃ x∈fset (insert-before v e y xs). ⋃ (dverts ‘
Basic-BNFs.fsts x))
      using t1-def by simp
    have 1: dverts (Node v {|(t1,e1)|}) = insert v (dverts t1) by simp
    show ?thesis
    proof(cases ∃ t. t ∈ fst ‘ fset xs ∧ root t = y)
      case True
      then show ?thesis using t1-def 0 insert.IH by simp
    next
      case False
      then show ?thesis using t1-def 0 insert-before-not-y-id by force
    qed
  next
    case False
    then have 0: ∃ t. t ∈ fst ‘ fset xs ∧ root t = y using insert.prems t1-def by
force
    have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
      by (simp add: insert.hyps prod.case-distrib)
    then have insert-before v e y (finsert x xs) = finsert x (insert-before v e y xs)
      by (simp add: False t1-def)
    then show ?thesis using insert.IH insert.prems 0 by simp
  qed
```

**qed**

**lemma** *insert-between-add-v-if-x-in*:
*x* ∈ *dverts t* ⟹ *dverts* (*insert-between v e x y t*) = *insert v* (*dverts t*)
**using** *wf-verts* **proof**(*induction t*)
**case** (*Node r xs*)
**show** *?case*
**proof**(*cases x=r*)
**case** *False*
**then obtain** *t e1* **where** *t-def*: (*t*,*e1*) ∈ *fset xs x* ∈ *dverts t* **using** *Node.prems*(*1*)
**by** *auto*
**then have** ∀(*t2*,*e2*) ∈ *fset xs*. (*t*,*e1*) ≠ (*t2*,*e2*) ⟶ *x* ∉ *dverts t2*
**using** *Node.prems*(*2*) **by** (*fastforce simp*: *wf-dverts-iff-dverts'*)
**then have** ∀(*t2*,*e2*) ∈ *fset xs*. (*t*,*e1*) = (*t2*,*e2*) ∨ (*insert-between v e x y t2*)
= *t2*
**using** *insert-between-id-if-notin* **by** *fast*
**moreover have** (*insert-between v e x y t*,*e1*)
∈ *fset* ((λ(*t*,*e1*). (*insert-between v e x y t*,*e1*)) |`| *xs*) **using** *t-def*(*1*) **by** *force*
**moreover have** *dverts* (*insert-between v e x y t*) = *insert v* (*dverts t*)
**using** *Node.IH Node.prems*(*2*) *t-def* **by** *auto*
**ultimately show** *?thesis* **using** *False* **by** *force*
**qed** (*auto simp*: *dverts-insert-before-aux*)
**qed**

**lemma** *insert-before-only1-new*:
**assumes** ∀(*x*,*e1*) ∈ *fset xs*. ∀(*y*,*e2*) ∈ *fset xs*. (*dverts x* ∩ *dverts y* = {} ∨
(*x*,*e1*)=(*y*,*e2*))
**and** (*t1*,*e1*) ≠ (*t2*,*e2*)
**and** (*t1*,*e1*) ∈ *fset* (*insert-before v e y xs*)
**and** (*t2*,*e2*) ∈ *fset* (*insert-before v e y xs*)
**shows** (*t1*,*e1*) ∈ *fset xs* ∨ (*t2*,*e2*) ∈ *fset xs*
**proof** (*rule ccontr*)
**assume** ¬((*t1*,*e1*) ∈ *fset xs* ∨ (*t2*,*e2*) ∈ *fset xs*)
**then have** *asm*: (*t1*,*e1*) ∉ *fset xs* (*t2*,*e2*) ∉ *fset xs* **by** *auto*
**obtain** *t3 e3* **where** *t3-def*: (*t3*, *e3*)∈*fset xs Node v* {|(*t3*, *e3*)|} = *t1 root t3* =
*y e1*=*e*
**using** *in-insert-before-child-in-orig assms*(*3*) *asm*(*1*) **by** *fast*
**obtain** *t4 e4* **where** *t4-def*: (*t4*, *e4*)∈*fset xs Node v* {|(*t4*, *e4*)|} = *t2 root t4* =
*y e2*=*e*
**using** *in-insert-before-child-in-orig assms*(*4*) *asm*(*2*) **by** *fast*
**then have** *dverts t3* ∩ *dverts t4* ≠ {} **using** *t3-def*(*3*) *dtree.set-sel*(*1*) **by** *force*
**then have** (*t3*,*e3*) = (*t4*,*e4*) **using** *assms*(*1*) *t3-def*(*1*) *t4-def*(*1*) **by** *fast*
**then show** *False* **using** *assms*(*2*) *t3-def*(*2*,*4*) *t4-def*(*2*,*4*) **by** *fast*
**qed**

**lemma** *disjoint-dverts-aux1*:
**assumes** ∀(*t1*,*e1*) ∈ *fset xs*. ∀(*t2*,*e2*) ∈ *fset xs*. (*dverts t1* ∩ *dverts t2* = {} ∨
(*t1*,*e1*)=(*t2*,*e2*))
**and** *v* ∉ *dverts* (*Node r xs*)

**and** $(t1,e1) \in fset\ (insert\text{-}before\ v\ e\ y\ xs)$
**and** $(t2,e2) \in fset\ (insert\text{-}before\ v\ e\ y\ xs)$
**and** $(t1,e1) \neq (t2,e2)$
**shows** $dverts\ t1 \cap dverts\ t2 = \{\}$
**proof** −
**consider** $(t1,e1) \in fset\ xs\ (t2,e2) \in fset\ xs$
| $(t1,e1) \notin fset\ xs\ (t2,e2) \in fset\ xs$
| $(t1,e1) \in fset\ xs\ (t2,e2) \notin fset\ xs$
**using** *insert-before-only1-new assms*(1,3−5) **by** *fast*
**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then show** *?thesis* **using** *assms*(1,5) **by** *fast*
**next**
**case** *2*
**obtain** *t3 e3* **where** *t3-def*: $(t3,\ e3) \in fset\ xs\ Node\ v\ \{|(t3,\ e3)|\} = t1\ root\ t3 = y\ e1{=}e$
**using** *in-insert-before-child-in-orig assms*(3) *2* **by** *fast*
**then have** $y{\neq}v$ **using** *assms*(2) *dtree.set-sel*(1) **by** *force*
**then have** $(t3,e3) \neq (t2,e2)$ **using** *assms*(4) *t3-def*(3) *root-noty-if-in-insert-before* **by** *fast*
**then have** $dverts\ t3 \cap dverts\ t2 = \{\}$ **using** *assms*(1) *2*(2) *t3-def*(1) **by** *fast*
**then show** *?thesis* **using** *assms*(1,2) *t3-def*(1,2) *2*(2) **by** *force*
**next**
**case** *3*
**obtain** *t3 e3* **where** *t3-def*: $(t3,\ e3) \in fset\ xs\ Node\ v\ \{|(t3,\ e3)|\} = t2\ root\ t3 = y\ e2{=}e$
**using** *in-insert-before-child-in-orig assms*(4) *3* **by** *fast*
**then have** $y{\neq}v$ **using** *assms*(2) *dtree.set-sel*(1) **by** *force*
**then have** $(t3,e3) \neq (t1,e1)$ **using** *assms*(3) *t3-def*(3) *root-noty-if-in-insert-before* **by** *fast*
**then have** $dverts\ t3 \cap dverts\ t1 = \{\}$ **using** *assms*(1) *3*(1) *t3-def*(1) **by** *fast*
**then show** *?thesis* **using** *assms*(2) *t3-def*(2) *3*(1) **by** *force*
**qed**
**qed**

**lemma** *disjoint-dverts-aux1′*:
**assumes** *wf-dverts* $(Node\ r\ xs)$ **and** $v \notin dverts\ (Node\ r\ xs)$
**shows** $\forall (x,e1) \in fset\ (insert\text{-}before\ v\ e\ y\ xs).\ \forall (y,e2) \in fset\ (insert\text{-}before\ v\ e\ y\ xs).$
$dverts\ x \cap dverts\ y = \{\} \vee (x,e1) = (y,e2)$
**using** *assms disjoint-dverts-aux1 disjoint-dverts-if-wf* **unfolding** *wf-dverts-iff-dverts′* **by** *fast*

**lemma** *insert-before-wf-dverts*:
$[\![\forall (t,e1) \in fset\ xs.\ wf\text{-}dverts\ t;\ v \notin dverts(Node\ r\ xs);\ (t1,e1) \in fset\ (insert\text{-}before\ v\ e\ y\ xs)]\!]$
$\Longrightarrow$ *wf-dverts t1*
**proof**(*induction xs*)

```
  case (insert x xs)
  let ?f = (λ(t1,e1). if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1))
  show ?case
  proof(cases (t1,e1) ∈ fset (insert-before v e y xs))
    case in-xs: True
    then show ?thesis
    proof(cases ?f x = (t1,e1))
      case True
      have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
        by (simp add: insert.hyps prod.case-distrib)
      then have insert-before v e y (finsert x xs) = insert-before v e y xs
        using True in-xs by fastforce
      then show ?thesis using insert.IH insert.prems by simp
    next
      case False
      then show ?thesis using in-xs insert.IH insert.prems(1,2) by auto
    qed
  next
    case False
    have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
      by (simp add: insert.hyps prod.case-distrib)
    then have ?f x = (t1,e1) using False insert.prems(3) by fastforce
    then show ?thesis
    proof(cases root t1 = v)
      case True
     then have (t1,e1) ∉ fset (finsert x xs) using insert.prems(2) dtree.set-sel(1)
by force
      then obtain t2 e2 where
       t2-def: (t2, e2)∈fset (finsert x xs) Node v {|(t2, e2)|} = t1 root t2 = y e1=e
        using in-insert-before-child-in-orig[of t1] insert.prems(3) by blast
     then show ?thesis using insert.prems(1,2) by (fastforce simp: wf-dverts-iff-dverts')
    next
      case False
      then have (t1,e1) = x
        using insert.prems(1) dtree.sel(1) ‹?f x = (t1,e1)›
        by (smt (verit, ccfv-SIG) Pair-inject old.prod.case case-prodE finsertI1)
      then show ?thesis using insert.prems(1) by auto
    qed
  qed
qed (simp)

lemma insert-before-root-nin-verts:
  ⟦∀(t,e1)∈fset xs. r ∉ dverts t; v ∉ dverts (Node r xs); (t1,e1) ∈ fset (insert-before
v e y xs)⟧
    ⟹ r ∉ dverts t1
proof(induction xs)
  case (insert x xs)
  let ?f = (λ(t1,e1). if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1))
  show ?case
```

```
  proof(cases (t1,e1) ∈ fset (insert-before v e y xs))
    case in-xs: True
    then show ?thesis
    proof(cases ?f x = (t1,e1))
      case True
      have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
        by (simp add: insert.hyps prod.case-distrib)
      then have insert-before v e y (finsert x xs) = insert-before v e y xs
        using True in-xs by fastforce
      then show ?thesis using insert.IH insert.prems by simp
    next
      case False
      then show ?thesis using in-xs insert.IH insert.prems(1,2) by auto
    qed
  next
    case False
    have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
      by (simp add: insert.hyps prod.case-distrib)
    then have ?f x = (t1,e1) using False insert.prems(3) by fastforce
    then show ?thesis
    proof(cases root t1 = v)
      case True
     then have (t1,e1) ∉ fset (finsert x xs) using insert.prems(2) dtree.set-sel(1)
by force
      then obtain t2 e2 where
       t2-def: (t2, e2)∈fset (finsert x xs) Node v {|(t2, e2)|} = t1 root t2 = y e1=e
        using in-insert-before-child-in-orig[of t1] insert.prems(3) by blast
      then show ?thesis using insert.prems(1,2) by fastforce
    next
      case False
      then have (t1,e1) = x
        using insert.prems(1) dtree.sel(1) ‹?f x = (t1,e1)›
        by (smt (verit, ccfv-SIG) Pair-inject old.prod.case case-prodE finsertI1)
      then show ?thesis using insert.prems(1) by auto
    qed
  qed
qed (simp)

lemma disjoint-dverts-aux2:
  assumes wf-dverts (Node r xs) and v ∉ dverts (Node r xs)
  shows ∀(x,e1) ∈ fset (finsert (Node v {||},e) xs). ∀(y,e2) ∈ fset (finsert (Node
v {||},e) xs).
          dverts x ∩ dverts y = {} ∨ (x,e1) = (y,e2)
  using assms by (fastforce simp: wf-dverts-iff-dverts')

lemma disjoint-dverts-aux3:
  assumes (t2,e2) ∈ (λ(t1,e1). (insert-between v e x y t1, e1)) ‘ fset xs
      and (t3,e3) ∈ (λ(t1,e1). (insert-between v e x y t1, e1)) ‘ fset xs
      and (t2,e2)≠(t3,e3)
```

```
    and (t,e1) ∈ fset xs
    and x ∈ dverts t
    and wf-dverts (Node r xs)
    and v ∉ dverts (Node r xs)
  shows dverts t2 ∩ dverts t3 = {}
proof −
  have ∀(t2,e2) ∈ fset xs. (t,e1)=(t2,e2) ∨ x ∉ dverts t2
    using assms(4−6) by (fastforce simp: wf-dverts-iff-dverts')
  then have nt1-id: ∀(t2,e2) ∈ fset xs. (t,e1) = (t2,e2) ∨ insert-between v e x y
t2 = t2
    using insert-between-id-if-notin by fastforce
  have dverts-t1: dverts (insert-between v e x y t) = insert v (dverts t)
    using assms(5−6) by (simp add: insert-between-add-v-if-x-in)
  have t1-disj: ∀(t2,e2) ∈ fset xs. (t,e1) = (t2,e2) ∨ dverts t2 ∩ insert v (dverts
t) = {}
    using assms(4−7) by (fastforce simp: wf-dverts-iff-dverts')
  consider (t2,e2) = (insert-between v e x y t,e1)
      | (t3,e3) = (insert-between v e x y t,e1)
       | (t2,e2) ≠ (insert-between v e x y t,e1) (t3,e3) ≠ (insert-between v e x y
t,e1)
    by fast
  then show ?thesis
  proof(cases)
    case 1
    then have (t3,e3) ∈ fset xs using assms(2,3) nt1-id by fastforce
    moreover have (t3,e3) ≠ (t,e1) using assms(2,3) 1 nt1-id by fastforce
    ultimately show ?thesis using 1 t1-disj dverts-t1 by fastforce
  next
    case 2
    then have (t2,e2) ∈ fset xs using assms(1,3) nt1-id by fastforce
    moreover have (t2,e2) ≠ (t,e1) using assms(1,3) 2 nt1-id by auto
    ultimately show ?thesis using 2 t1-disj dverts-t1 by fastforce
  next
    case 3
    then have (t2,e2) ∈ fset xs using assms(1) nt1-id by fastforce
    moreover have (t3,e3) ∈ fset xs using assms(2) 3(2) nt1-id by auto
   ultimately show ?thesis using assms(3,6) by (fastforce simp: wf-dverts-iff-dverts')
  qed
qed

lemma insert-between-wf-dverts: v ∉ dverts t ⟹ wf-dverts (insert-between v e x
y t)
using wf-dtree-axioms proof(induction t)
  case (Node r xs)
  then interpret wf-dtree Node r xs by blast
  consider x=r ∃ t. t ∈ fst ‘ fset xs ∧ root t = y
           | x=r ¬(∃ t. t ∈ fst ‘ fset xs ∧ root t = y) | x≠r by fast
  then show ?case
  proof(cases)
```

```
    case 1
    then have insert-between v e x y (Node r xs) = Node r (insert-before v e y xs)
by simp
    moreover have ∀(x,e1) ∈ fset (insert-before v e y xs). r ∉ dverts x
      using insert-before-root-nin-verts wf-verts Node.prems(1)
      by (fastforce simp: wf-dverts-iff-dverts')
    moreover have ∀(x,e1) ∈ fset (insert-before v e y xs). wf-dverts x
      using insert-before-wf-dverts Node.prems(1) wf-verts by fastforce
    moreover have ∀(x, e1)∈fset (insert-before v e y xs).
          ∀(y, e2)∈fset (insert-before v e y xs). dverts x ∩ dverts y = {} ∨ (x, e1)
= (y, e2)
    using disjoint-dverts-aux1' Node.prems(1) wf-verts unfolding wf-dverts-iff-dverts'
by fast
    ultimately show ?thesis by (fastforce simp: wf-dverts-iff-dverts')
  next
    case 2
    then have insert-between v e x y (Node r xs) = Node r (finsert (Node v {||},e)
xs) by simp
    then show ?thesis
      using disjoint-dverts-aux2[of r xs v] Node.prems(1) wf-verts
      by (fastforce simp: wf-dverts-iff-dverts')
  next
    case 3
    let ?f = λ(t1,e1). (insert-between v e x y t1, e1)
    show ?thesis
    proof(cases ∃(t1,e1) ∈ fset xs. x ∈ dverts t1)
      case True
      then obtain t1 e1 where t1-def: (t1,e1) ∈ fset xs  x ∈ dverts t1 by blast
      then interpret T: wf-dtree t1 using wf-dtree-rec by blast
      have ∀(t2,e2) ∈ ?f ‘ fset xs. ∀(t3,e3) ∈ ?f ‘ fset xs.
            (t2,e2) = (t3,e3) ∨ dverts t2 ∩ dverts t3 = {}
        using T.disjoint-dverts-aux3 Node.prems(1) t1-def wf-verts by blast
      moreover have ⋀t2 e2. (t2,e2) ∈ ?f ‘ fset xs ⟶ r ∉ dverts t2 ∧ wf-dverts
t2
      proof
        fix t2 e2
        assume asm: (t2,e2) ∈ ?f ‘ fset xs
        then show r ∉ dverts t2 ∧ wf-dverts t2
        proof(cases (t2,e2) = (insert-between v e x y t1,e1))
          case True
          then have wf-dverts (insert-between v e x y t1)
            using Node.IH Node.prems(1) T.wf-dtree-axioms t1-def(1) by auto
          then show ?thesis
            using Node.prems(1) wf-verts True T.insert-between-add-v-if-x-in t1-def
            by (auto simp: wf-dverts-iff-dverts')
        next
          case False
          have ∀(t2,e2) ∈ fset xs. (t1,e1)=(t2,e2) ∨ x ∉ dverts t2
            using wf-verts t1-def by (fastforce simp: wf-dverts-iff-dverts')
```

```
        then have ∀(t2,e2) ∈ fset xs. (t1,e1) = (t2,e2) ∨ insert-between v e x y
t2 = t2
          using insert-between-id-if-notin by fastforce
       then show ?thesis using wf-verts asm False by (fastforce simp: wf-dverts-iff-dverts')
        qed
      qed
      ultimately show ?thesis using 3 by (fastforce simp: wf-dverts-iff-dverts')
    next
      case False
      then show ?thesis
        using wf-verts 3 insert-between-id-if-notin fst-conv
          by (smt (verit, ccfv-threshold) fsts.cases dtree.inject dtree.set-cases(1)
case-prodI2)
    qed
  qed
qed

lemma darcs-insert-before-aux:
  ∃t. t ∈ fst ‘ fset xs ∧ root t = y
    ⟹ (⋃x∈fset (insert-before v e y xs). ⋃ (darcs ‘ Basic-BNFs.fsts x) ∪ Ba-
sic-BNFs.snds x)
       = insert e (⋃x∈fset xs. ⋃ (darcs ‘ Basic-BNFs.fsts x) ∪ Basic-BNFs.snds
x)
proof(induction xs)
  case (insert x xs)
  let ?f = (λ(t1,e1). if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1))
  let ?xs = insert-before v e y (finsert x xs)
  obtain t1 e1 where t1-def: x = (t1,e1) by fastforce
  then show ?case
  proof(cases root t1 = y)
    case True
    then have ?xs = finsert (?f x) (insert-before v e y xs)
      by (simp add: insert.hyps prod.case-distrib)
    then have ?xs = finsert (Node v {|(t1,e1)|},e) (insert-before v e y xs)
      using t1-def True by simp
   then have 0: (⋃x∈fset ?xs. ⋃ (darcs ‘ Basic-BNFs.fsts x) ∪ Basic-BNFs.snds
x)
      = (⋃ (darcs ‘ {Node v {|(t1,e1)|}}) ∪ {e})
          ∪ (⋃x∈fset (insert-before v e y xs). ⋃ (darcs ‘ Basic-BNFs.fsts x) ∪
Basic-BNFs.snds x)
      using t1-def by simp
    have 1: dverts (Node v {|(t1,e1)|}) = insert v (dverts t1) by simp
    show ?thesis
    proof(cases ∃t. t ∈ fst ‘ fset xs ∧ root t = y)
      case True
      then show ?thesis using t1-def 0 insert.IH by simp
    next
      case False
      then show ?thesis using t1-def 0 insert-before-not-y-id by force
```

```
    qed
  next
    case False
    then have 0: ∃ t. t ∈ fst ‘ fset xs ∧ root t = y using insert.prems t1-def by
force
    have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
      by (simp add: insert.hyps prod.case-distrib)
    then have insert-before v e y (finsert x xs) = finsert x (insert-before v e y xs)
      by (simp add: False t1-def)
    then show ?thesis using insert.IH insert.prems 0 by simp
  qed
qed (simp)

lemma insert-between-add-e-if-x-in:
  x ∈ dverts t ⟹ darcs (insert-between v e x y t) = insert e (darcs t)
using wf-verts proof(induction t)
  case (Node r xs)
  show ?case
  proof(cases x=r)
    case False
   then obtain t e1 where t-def: (t,e1) ∈ fset xs x ∈ dverts t using Node.prems(1)
by auto
    then have ∀ (t2,e2) ∈ fset xs. (t,e1) ≠ (t2,e2) ⟶ x ∉ dverts t2
      using Node.prems(2) by (fastforce simp: wf-dverts-iff-dverts′)
    then have ∀ (t2,e2) ∈ fset xs. (t,e1) = (t2,e2) ∨ (insert-between v e x y t2)
= t2
      using insert-between-id-if-notin by fast
    moreover have (insert-between v e x y t,e1)
      ∈ fset ((λ(t,e1). (insert-between v e x y t,e1)) |‘| xs) using t-def(1) by force
    moreover have darcs (insert-between v e x y t) = insert e (darcs t)
      using Node.IH Node.prems(2) t-def by auto
    ultimately show ?thesis using False by force
  qed (auto simp: darcs-insert-before-aux)
qed

lemma disjoint-darcs-aux1-aux1:
  assumes disjoint-darcs xs
      and wf-dverts (Node r xs)
      and v ∉ dverts (Node r xs)
      and e ∉ darcs (Node r xs)
      and (t1,e1) ∈ fset (insert-before v e y xs)
      and (t2,e2) ∈ fset (insert-before v e y xs)
      and (t1,e1) ≠ (t2,e2)
    shows (darcs t1 ∪ {e1}) ∩ (darcs t2 ∪ {e2}) = {}
proof −
  consider (t1,e1) ∈ fset xs (t2,e2) ∈ fset xs
        | (t1,e1) ∉ fset xs (t2,e2) ∈ fset xs
        | (t1,e1) ∈ fset xs (t2,e2) ∉ fset xs
   using insert-before-only1-new assms(2,5−7) by (fastforce simp: wf-dverts-iff-dverts′)
```

**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then show** *?thesis* **using** *assms*(*1*,*7*) **by** *fast*
**next**
**case** *2*
**obtain** *t3 e3* **where** *t3-def*: (*t3*, *e3*)∈*fset xs Node v* {|(*t3*, *e3*)|} = *t1 root t3* = *y e1*=*e*
**using** *in-insert-before-child-in-orig assms*(*5*) *2* **by** *fast*
**then have** *v*≠*y* **using** *assms*(*3*) *dtree.set-sel*(*1*) **by** *force*
**then have** (*t3*,*e3*) ≠ (*t2*,*e2*) **using** *assms*(*6*) *t3-def*(*3*) *root-noty-if-in-insert-before* **by** *fast*
**then have** (*darcs t3* ∪ {*e3*}) ∩ (*darcs t2* ∪ {*e2*}) = {} **using** *assms*(*1*) *2*(*2*) *t3-def*(*1*) **by** *fast*
**then show** *?thesis* **using** *assms*(*4*) *t3-def*(*4*) *2*(*2*) *t3-def*(*2*) **by** *force*
**next**
**case** *3*
**obtain** *t3 e3* **where** *t3-def*: (*t3*, *e3*)∈*fset xs Node v* {|(*t3*, *e3*)|} = *t2 root t3* = *y e2*=*e*
**using** *in-insert-before-child-in-orig assms*(*6*) *3* **by** *fast*
**then have** *v*≠*y* **using** *assms*(*3*) *dtree.set-sel*(*1*) **by** *force*
**then have** (*t3*,*e3*) ≠ (*t1*,*e1*) **using** *assms*(*5*) *t3-def*(*3*) *root-noty-if-in-insert-before* **by** *fast*
**then have** (*darcs t3* ∪ {*e3*}) ∩ (*darcs t1* ∪ {*e1*}) = {} **using** *assms*(*1*) *3*(*1*) *t3-def*(*1*) **by** *fast*
**then show** *?thesis* **using** *assms*(*4*) *t3-def*(*4*) *3*(*1*) *t3-def*(*2*) **by** *force*
**qed**
**qed**

**lemma** *disjoint-darcs-aux1-aux2*:
**assumes** *disjoint-darcs xs*
**and** *e* ∉ *darcs* (*Node r xs*)
**and** (*t1*,*e1*) ∈ *fset* (*insert-before v e y xs*)
**shows** *e1* ∉ *darcs t1*
**proof**(*cases* (*t1*,*e1*) ∈ *fset xs*)
**case** *True*
**then show** *?thesis* **using** *assms*(*1*) **by** *fast*
**next**
**case** *False*
**then obtain** *t3 e3* **where** (*t3*, *e3*)∈*fset xs Node v* {|(*t3*, *e3*)|} = *t1 e1*=*e*
**using** *in-insert-before-child-in-orig assms*(*3*) **by** *fast*
**then show** *?thesis* **using** *assms*(*2*) **by** *auto*
**qed**

**lemma** *disjoint-darcs-aux1*:
**assumes** *wf-dverts* (*Node r xs*) **and** *v* ∉ *dverts* (*Node r xs*)
**and** *wf-darcs* (*Node r xs*) **and** *e* ∉ *darcs* (*Node r xs*)
**shows** *disjoint-darcs* (*insert-before v e y xs*) (**is** *disjoint-darcs ?xs*)
**proof** −

```
  have 0: disjoint-darcs xs using assms(3) disjoint-darcs-if-wf-xs by simp
  then have ∀(t1,e1) ∈ fset ?xs. e1 ∉ darcs t1
    using disjoint-darcs-aux1-aux2[of xs] assms(4) by fast
  moreover have ∀(t1,e1) ∈ fset ?xs. ∀(t2,e2) ∈ fset ?xs.
            (darcs t1 ∪ {e1}) ∩ (darcs t2 ∪ {e2}) = {} ∨ (t1,e1) = (t2,e2)
    using disjoint-darcs-aux1-aux1[of xs] assms(1,2,4) 0 by blast
  ultimately show ?thesis by fast
qed

lemma insert-before-wf-darcs:
  ⟦wf-darcs (Node r xs); e ∉ darcs (Node r xs); (t1,e1) ∈ fset (insert-before v e y
xs)⟧
    ⟹ wf-darcs t1
proof(induction xs)
  case (insert x xs)
  let ?f = (λ(t1,e1). if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1))
  show ?case
  proof(cases (t1,e1) ∈ fset (insert-before v e y xs))
    case in-xs: True
    then show ?thesis
    proof(cases ?f x = (t1,e1))
      case True
      have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
        by (simp add: insert.hyps prod.case-distrib)
      then have insert-before v e y (finsert x xs) = insert-before v e y xs
        using True in-xs by fastforce
      moreover have disjoint-darcs xs
       using disjoint-darcs-insert[OF disjoint-darcs-if-wf-xs[OF insert.prems(1)]] .
      ultimately show ?thesis
        using insert.IH insert.prems unfolding wf-darcs-iff-darcs' by force
    next
      case False
      have disjoint-darcs xs
       using disjoint-darcs-insert[OF disjoint-darcs-if-wf-xs[OF insert.prems(1)]] .
      then show ?thesis
     using in-xs False insert.IH insert.prems(1,2) by (simp add: wf-darcs-iff-darcs')
    qed
  next
    case False
    have insert-before v e y (finsert x xs) = finsert (?f x) (insert-before v e y xs)
      by (simp add: insert.hyps prod.case-distrib)
    then have 0: ?f x = (t1,e1) using False insert.prems(3) by fastforce
    then show ?thesis
    proof(cases e1=e)
      case True
     then have (t1,e1) ∉ fset (finsert x xs) using insert.prems(2) dtree.set-sel(1)
by force
      then obtain t2 e2 where
       t2-def: (t2, e2)∈fset (finsert x xs) Node v {|(t2, e2)|} = t1 root t2 = y e1=e
```

```
      using in-insert-before-child-in-orig[of t1] insert.prems(3) by blast
    then show ?thesis
      using insert.prems(1) t2-def by (fastforce simp: wf-darcs-iff-darcs')
  next
    case False
    then have (t1,e1) = x
    by (smt (z3) 0 old.prod.exhaust prod.inject case-prod-Pair-iden case-prod-conv)
    then show ?thesis using insert.prems(1) by auto
  qed
 qed
qed (simp)

lemma disjoint-darcs-aux2:
 assumes wf-darcs (Node r xs) and e ∉ darcs (Node r xs)
 shows disjoint-darcs (finsert (Node v {||},e) xs)
 using assms unfolding wf-darcs-iff-darcs' by fastforce

lemma disjoint-darcs-aux3-aux1:
 assumes (t,e1) ∈ fset xs
    and x ∈ dverts t
    and wf-darcs (Node r xs)
    and e ∉ darcs (Node r xs)
    and (t2,e2) ∈ (λ(t1,e1). (insert-between v e x y t1, e1)) ‘ fset xs
    and (t3,e3) ∈ (λ(t1,e1). (insert-between v e x y t1, e1)) ‘ fset xs
    and (t2,e2)≠(t3,e3)
    and wf-dverts (Node r xs)
  shows (darcs t2 ∪ {e2}) ∩ (darcs t3 ∪ {e3}) = {}
proof −
 have ∀(t2,e2) ∈ fset xs. (t,e1)=(t2,e2) ∨ x ∉ dverts t2
  using assms(1,2,8) by (fastforce simp: wf-dverts-iff-dverts')
 then have nt1-id: ∀(t2,e2) ∈ fset xs. (t,e1) = (t2,e2) ∨ insert-between v e x y
t2 = t2
  using insert-between-id-if-notin by fastforce
 have darcs-t: darcs (insert-between v e x y t) = insert e (darcs t)
  using assms(2,3) by (simp add: insert-between-add-e-if-x-in)
 consider (t2,e2) = (insert-between v e x y t,e1)
    | (t3,e3) = (insert-between v e x y t,e1)
     | (t2,e2) ≠ (insert-between v e x y t,e1) (t3,e3) ≠ (insert-between v e x y
t,e1)
  by fast
 then show ?thesis
 proof(cases)
  case 1
  then have (t3,e3) ∈ fset xs using assms(6,7) nt1-id by fastforce
  moreover have (t3,e3) ≠ (t,e1) using assms(6,7) 1 nt1-id by fastforce
  ultimately have (darcs t ∪ {e1,e}) ∩ (darcs t3 ∪ {e3}) = {}
   using assms(1,3,4) unfolding wf-darcs-iff-darcs' by fastforce
  then show ?thesis using 1 darcs-t by auto
 next
```

```
    case 2
    then have (t2,e2) ∈ fset xs using assms(5,7) nt1-id by fastforce
    moreover have (t2,e2) ≠ (t,e1) using assms(5,7) 2 nt1-id by auto
    ultimately have (darcs t ∪ {e1,e}) ∩ (darcs t2 ∪ {e2}) = {}
      using assms(1,3,4) unfolding wf-darcs-iff-darcs' by fastforce
    then show ?thesis using 2 darcs-t by force
  next
    case 3
    then have (t2,e2) ∈ fset xs using assms(5) nt1-id by fastforce
    moreover have (t3,e3) ∈ fset xs using assms(6) 3(2) nt1-id by auto
    ultimately show ?thesis using assms(3,7) unfolding wf-darcs-iff-darcs' by
fastforce
  qed
qed

lemma disjoint-darcs-aux3-aux2:
  assumes (t,e1) ∈ fset xs
      and x ∈ dverts t
      and wf-darcs (Node r xs)
      and e ∉ darcs (Node r xs)
      and (t2,e2) ∈ (λ(t1,e1). (insert-between v e x y t1, e1)) ‘ fset xs
      and wf-dverts (Node r xs)
    shows e2 ∉ darcs t2
proof(cases (t2,e2) ∈ fset xs)
  case True
  then show ?thesis using assms(3) unfolding wf-darcs-iff-darcs' by auto
next
  case False
  obtain t1 where t1-def: insert-between v e x y t1 = t2 (t1,e2) ∈ fset xs
    using assms(5) by fast
  then have x ∈ dverts t1 using insert-between-id-if-notin False by fastforce
 then have t = t1 using assms(1,2,6) t1-def(2) by (fastforce simp: wf-dverts-iff-dverts')
  then have darcs-t: darcs t2 = insert e (darcs t1)
    using insert-between-add-e-if-x-in assms(2) t1-def(1) by force
 then show ?thesis using assms(3,4) t1-def(2) unfolding wf-darcs-iff-darcs' by
fastforce
qed

lemma disjoint-darcs-aux3:
  assumes (t,e1) ∈ fset xs
      and x ∈ dverts t
      and wf-darcs (Node r xs)
      and e ∉ darcs (Node r xs)
      and wf-dverts (Node r xs)
    shows disjoint-darcs ((λ(t1,e1). (insert-between v e x y t1, e1)) |‘| xs)
proof −
  let ?xs = (λ(t1,e1). (insert-between v e x y t1, e1)) |‘| xs
  let ?xs' = (λ(t1,e1). (insert-between v e x y t1, e1)) ‘ fset xs
  have 0: fset ?xs = ?xs' by simp
```

```
  then have ∀(t1,e1) ∈ fset ?xs. e1 ∉ darcs t1
    using disjoint-darcs-aux3-aux2 assms by blast
  moreover have ∀(t1,e1) ∈ ?xs'. ∀(t2,e2) ∈ ?xs'.
            (darcs t1 ∪ {e1}) ∩ (darcs t2 ∪ {e2}) = {} ∨ (t1,e1) = (t2,e2)
    using disjoint-darcs-aux3-aux1 assms by blast
  ultimately show ?thesis using 0 by fastforce
qed

lemma insert-between-wf-darcs:
  ⟦e ∉ darcs t; v ∉ dverts t ⟧ ⟹ wf-darcs (insert-between v e x y t)
using wf-dtree-axioms proof(induction t)
  case (Node r xs)
  then interpret wf-dtree Node r xs by blast
  consider x=r ∃ t. t ∈ fst ‘ fset xs ∧ root t = y
           | x=r ¬(∃ t. t ∈ fst ‘ fset xs ∧ root t = y) | x≠r by fast
  then show ?case
  proof(cases)
    case 1
    then have insert-between v e x y (Node r xs) = Node r (insert-before v e y xs)
by simp
    moreover have ∀(x,e1) ∈ fset (insert-before v e y xs). wf-darcs x
      using insert-before-wf-darcs Node.prems(1) wf-arcs by fast
    moreover have disjoint-darcs (insert-before v e y xs)
     using disjoint-darcs-aux1[OF wf-verts Node.prems(2) wf-arcs Node.prems(1)]
.
    ultimately show ?thesis by (simp add: wf-darcs-if-darcs'-aux)
  next
    case 2
    then have insert-between v e x y (Node r xs) = Node r (finsert (Node v {||},e)
xs) by simp
    then show ?thesis
    using disjoint-darcs-aux2 Node.prems(1) wf-arcs by (simp add: wf-darcs-iff-darcs')
  next
    case 3
    let ?f = λ(t1,e1). (insert-between v e x y t1, e1)
    show ?thesis
    proof(cases ∃(t1,e1) ∈ fset xs. x ∈ dverts t1)
      case True
      then obtain t1 e1 where t1-def: (t1,e1) ∈ fset xs  x ∈ dverts t1 by blast
      then interpret T: wf-dtree t1 using wf-dtree-rec by blast
      have ⋀t2 e2. (t2,e2) ∈ fset (?f |‘| xs) ⟶ wf-darcs t2
      proof
        fix t2 e2
        assume asm: (t2,e2) ∈ fset (?f |‘| xs)
        then show wf-darcs t2
        proof(cases (t2,e2) = (insert-between v e x y t1,e1))
          case True
          then have wf-darcs (insert-between v e x y t1)
            using Node t1-def(1) T.wf-dtree-axioms
```

```
      by (metis dtree.set-intros(2) dtree.set-intros(3) insertI1 prod-set-simps(1))
      then show ?thesis using True by blast
    next
      case False
      have ∀(t2,e2) ∈ fset xs. (t1,e1)=(t2,e2) ∨ x ∉ dverts t2
        using wf-verts t1-def by (fastforce simp: wf-dverts-iff-dverts')
      then have ∀(t2,e2) ∈ fset xs. (t1,e1) = (t2,e2) ∨ insert-between v e x y
t2 = t2
        using insert-between-id-if-notin by fastforce
      then show ?thesis using wf-arcs asm False by fastforce
    qed
  qed
  moreover have disjoint-darcs (?f |`| xs)
    using T.disjoint-darcs-aux3 Node.prems(1) t1-def wf-arcs wf-verts by pres-
burger
  ultimately show ?thesis using 3 by (fastforce simp: wf-darcs-iff-darcs')
next
  case False
  then show ?thesis
    using wf-arcs 3 insert-between-id-if-notin fst-conv
      by (smt (verit, ccfv-threshold) fsts.cases dtree.inject dtree.set-cases(1)
case-prodI2)
qed
qed

theorem insert-between-wf-dtree:
  ⟦e ∉ darcs t; v ∉ dverts t ⟧ ⟹ wf-dtree (insert-between v e x y t)
  by (simp add: insert-between-wf-dverts insert-between-wf-darcs wf-dtree-def)

lemma snds-neq-card-eq-card-snd:
  ∀(t,e) ∈ fset xs. ∀(t2,e2) ∈ fset xs. e≠e2 ∨ (t,e) = (t2,e2) ⟹ fcard xs = fcard
(snd |`| xs)
proof(induction xs)
  case empty
  then have (snd |`| {||}) = {||} by blast
  then show ?case by (simp add: fcard-fempty)
next
  case (insert x xs)
  have fcard xs = fcard (snd |`| xs) using insert.IH insert.prems by fastforce
  moreover have snd x |∉| snd |`| xs
  proof
    assume asm: snd x |∈| snd |`| xs
    then obtain t e where t-def: x = (t,e) by fastforce
    then obtain t2 where t2-def: (t2,e) |∈| xs using asm by auto
    then have (t,e)≠(t2,e) using insert.hyps t-def by blast
    moreover have (t,e) ∈ fset (finsert x xs) using t-def by simp
    moreover have (t2,e) ∈ fset (finsert x xs) using t2-def by fastforce
    ultimately show False using insert.prems by fast
```

**qed**
**ultimately show** *?case* **by** (*simp add*: *fcard-finsert-disjoint local.insert.hyps*)
**qed**

**lemma** *snds-neq-img-snds-neq*:
**assumes** $\forall (t,e) \in fset\ xs.\ \forall (t2,e2) \in fset\ xs.\ e \neq e2 \vee (t,e) = (t2,e2)$
**shows** $\forall (t1,e1) \in fset\ ((\lambda(t1,e1).\ (f\ t1,\ e1))\ |`|\ xs).$
$\forall (t2,e2) \in fset\ ((\lambda(t1,e1).\ (f\ t1,\ e1))\ |`|\ xs).\ e1 \neq e2 \vee (t1,e1) = (t2,e2)$
**using** *assms* **by** *auto*

**lemma** *snds-neq-if-disjoint-darcs*:
**assumes** *disjoint-darcs xs*
**shows** $\forall (t,e) \in fset\ xs.\ \forall (t2,e2) \in fset\ xs.\ e \neq e2 \vee (t,e) = (t2,e2)$
**using** *assms* **by** *fast*

**lemma** *snds-neq-img-card-eq*:
**assumes** $\forall (t,e) \in fset\ xs.\ \forall (t2,e2) \in fset\ xs.\ e \neq e2 \vee (t,e) = (t2,e2)$
**shows** $fcard\ ((\lambda(t1,e1).\ (f\ t1,\ e1))\ |`|\ xs) = fcard\ xs$
**proof** −
**let** $?f = \lambda(t1,e1).\ (f\ t1,\ e1)$
**have** $\forall (t,e) \in fset\ (?f\ |`|\ xs).\ \forall (t2,e2) \in fset\ (?f\ |`|\ xs).\ e \neq e2 \vee (t,e) = (t2,e2)$
**using** *assms snds-neq-img-snds-neq* **by** *auto*
**then have** $fcard\ (?f\ |`|\ xs) = fcard\ (snd\ |`|\ (?f\ |`|\ xs))$
**using** *snds-neq-card-eq-card-snd* **by** *blast*
**moreover have** $snd\ |`|\ (?f\ |`|\ xs) = snd\ |`|\ xs$ **by** *force*
**moreover have** $fcard\ xs = fcard\ (snd\ |`|\ xs)$ **using** *snds-neq-card-eq-card-snd assms* **by** *blast*
**ultimately show** *?thesis* **by** *simp*
**qed**

**lemma** *fst-neq-img-card-eq*:
**assumes** $\forall (t,e) \in fset\ xs.\ \forall (t2,e2) \in fset\ xs.\ f\ t \neq f\ t2 \vee (t,e) = (t2,e2)$
**shows** $fcard\ ((\lambda(t1,e1).\ (f\ t1,\ e1))\ |`|\ xs) = fcard\ xs$
**using** *assms* **proof**(*induction xs*)
**case** *empty*
**then have** $(snd\ |`|\ \{||\}) = \{||\}$ **by** *blast*
**then show** *?case* **by** (*simp add*: *fcard-fempty*)
**next**
**case** (*insert x xs*)
**have** $fcard\ xs = fcard\ ((\lambda(t1,e1).\ (f\ t1,\ e1))\ |`|\ xs)$ **using** *insert* **by** *fastforce*
**moreover have** $(\lambda(t1,e1).\ (f\ t1,\ e1))\ x\ |\notin|\ (\lambda(t1,e1).\ (f\ t1,\ e1))\ |`|\ xs$
**proof**
**assume** *asm*: $(\lambda(t1,e1).\ (f\ t1,\ e1))\ x\ |\in|\ (\lambda(t1,e1).\ (f\ t1,\ e1))\ |`|\ xs$
**then obtain** *t e* **where** *t-def*: $x = (t,e)$ **by** *fastforce*
**then obtain** *t2 e2* **where** *t2-def*:
$(t2,e2)\ |\in|\ xs\ (\lambda(t1,e1).\ (f\ t1,\ e1))\ (t2,e2) = (\lambda(t1,e1).\ (f\ t1,\ e1))\ (t,e)$
**using** *asm* **by** *auto*
**then have** $(t,e) \neq (t2,e)$ **using** *insert.hyps t-def* **by** *fast*
**moreover have** $(t,e) \in fset\ (finsert\ x\ xs)$ **using** *t-def* **by** *simp*

```
    moreover have (t2,e2) ∈ fset (finsert x xs) using t2-def(1) by fastforce
    ultimately show False using insert.prems t2-def(2) by fast
  qed
  ultimately show ?case by (simp add: fcard-finsert-disjoint local.insert.hyps)
qed

lemma x-notin-insert-before:
  assumes x |∉| xs and wf-dverts (Node r (finsert x xs))
  shows (λ(t1,e1). if root t1 = y then (Node v {|(t1,e1)|},e) else (t1,e1)) x
        |∉| (insert-before v e y xs) (is ?f x |∉|-)
proof (cases root (fst x) = y)
  case True
  then obtain t1 e1 where t1-def: x = (t1,e1) root t1 = y by fastforce
  then have 0: ∀(t2,e2) ∈ fset xs. dverts t1 ∩ dverts t2 = {}
    using assms disjoint-dverts-if-wf-aux by fastforce
  then have ∀(t2,e2) ∈ fset xs. root t2 ≠ y
    by (smt (verit, del-insts) dtree.set-sel(1) t1-def(2) case-prodD case-prodI2 dis-
joint-iff)
  hence ∄t. t ∈ fst ‘ fset xs ∧ dtree.root t = y
    by fastforce
  then have 1: (insert-before v e y xs) = xs using insert-before-not-y-id by fast-
force
  have ?f x = (Node v {|(t1,e1)|},e) using t1-def by simp
  then have ∀(t2,e2) ∈ fset xs. (fst (?f x)) ≠ t2 using 0 dtree.set-sel(1) by
fastforce
  then have ∀(t2,e2) ∈ fset (insert-before v e y xs). ?f x ≠ (t2,e2) using 1 by
fastforce
  then show ?thesis by fast
next
  case False
 then have x-id: ?f x = x by (smt (verit) old.prod.exhaust case-prod-conv fst-conv)
  then show ?thesis
  proof(cases ∃t1. t1 ∈ fst ‘ fset xs ∧ root t1 = y)
    case True
    then obtain t1 e1 where t1-def: (t1,e1) ∈ fset xs root t1 = y by force
    then have (t1,e1) ∈ fset (finsert x xs) by auto
    then have 0: ∀(t2,e2) ∈ fset (finsert x xs). (t1,e1) = (t2,e2) ∨ dverts t1 ∩
dverts t2 = {}
      using assms(2) disjoint-dverts-if-wf-aux by fast
    then have ∀(t2,e2) ∈ fset (finsert x xs). (t1,e1) = (t2,e2) ∨ root t2 ≠ y
      using dtree.set-sel(1) t1-def(2) insert-not-empty
     by (smt (verit, ccfv-threshold) Int-insert-right-if1 prod.case-eq-if insert-absorb)
    then have ∄t. t ∈ fst ‘ fset (xs |−| {|(t1,e1)|}) ∧ root t = y by fastforce
    then have 1: ?f |‘| (xs |−| {|(t1,e1)|}) = (xs |−| {|(t1,e1)|})
    using insert-before-not-y-id[of xs |−| {|(t1,e1)|}] by (simp add: insert-before-alt)
    have ?f (t1,e1) = (Node v {|(t1,e1)|},e) using t1-def by simp
   then have ?f |‘| xs = finsert (Node v {|(t1,e1)|},e) (?f |‘| (xs |−| {|(t1,e1)|}))
      using t1-def(1) by (metis (no-types, lifting) fimage-finsert finsert-fminus)
    then have ?f |‘| xs = finsert (Node v {|(t1,e1)|},e) (xs |−| {|(t1,e1)|})
```

**using** *1* **by** *simp*
**then have** *2*: *insert-before v e y xs = finsert (Node v {|(t1,e1)|},e) (xs |−| {|(t1,e1)|})*
**by** (*simp add*: *insert-before-alt*)
**have** *dverts t1 ∩ dverts (fst x) = {}* **using** *0 assms(1) t1-def(1)* **by** *fastforce*
**then have** *(Node v {|(t1,e1)|},e) ≠ x* **using** *dtree.set-sel(1)* **by** *fastforce*
**then show** *?thesis* **using** *2 assms(1) x-id* **by** *auto*
**next**
**case** *False*
**then have** *(insert-before v e y xs) = xs* **using** *insert-before-not-y-id* **by** *fastforce*
**then show** *?thesis* **using** *assms(1) x-id* **by** *simp*
**qed**
**qed**

**end**

**end**

**theory** *List-Dtree*
**imports** *Complex-Main Graph-Additions Dtree*
**begin**

# 8 Dtrees of Lists

## 8.1 Functions

**abbreviation** *remove-child* :: *'a ⇒ (('a,'b) dtree × 'b) fset ⇒ (('a,'b) dtree × 'b) fset* **where**
*remove-child x xs ≡ ffilter (λ(t,e). root t ≠ x) xs*

**abbreviation** *child2* ::
*'a ⇒ (('a,'b) dtree × 'b) fset ⇒ (('a,'b) dtree × 'b) fset ⇒ (('a,'b) dtree × 'b) fset* **where**
*child2 x zs xs ≡ ffold (λ(t,-) b. case t of Node r ys ⇒ if r = x then ys |∪| b else b) zs xs*

Combine children sets to a single set and append element to list.

**fun** *combine* :: *'a list ⇒ 'a list ⇒ ('a list,'b) dtree ⇒ ('a list,'b) dtree* **where**
*combine x y (Node r xs) = (if x=r ∧ (∃ t. t ∈ fst ‘ fset xs ∧ root t = y)*
*then Node (r@y) (child2 y (remove-child y xs) xs)*
*else Node r ((λ(t,e). (combine x y t,e)) |‘| xs))*

Basic *wf-dverts* property is not strong enough to be preserved in combine operation.

**fun** *dlverts* :: *('a list,'b) dtree ⇒ 'a set* **where**
*dlverts (Node r xs) = set r ∪ (⋃x∈fset xs. dlverts (fst x))*

**abbreviation** *disjoint-dlverts* :: *(('a list, 'b) dtree × 'b) fset ⇒ bool* **where**

*disjoint-dlverts xs* ≡
(∀ (*x*,*e1*) ∈ *fset xs*. ∀ (*y*,*e2*) ∈ *fset xs*. *dlverts x* ∩ *dlverts y* = {} ∨ (*x*,*e1*)=(*y*,*e2*))

**fun** *wf-dlverts* :: ('*a list*,'*b*) *dtree* ⇒ *bool* **where**
*wf-dlverts* (*Node r xs*) =
(*r* ≠ [] ∧ (∀ (*x*,*e1*) ∈ *fset xs*. *set r* ∩ *dlverts x* = {} ∧ *wf-dlverts x*) ∧ *disjoint-dlverts xs*)

**definition** *wf-dlverts'* :: ('*a list*,'*b*) *dtree* ⇒ *bool* **where**
*wf-dlverts' t* ⟷
*wf-dverts t* ∧ [] ∉ *dverts t* ∧ (∀ *v1*∈*dverts t*. ∀ *v2*∈*dverts t*. *set v1* ∩ *set v2* = {} ∨ *v1*=*v2*)

**fun** *wf-list-lverts* :: ('*a list*×'*b*) *list* ⇒ *bool* **where**
*wf-list-lverts* [] = *True*
| *wf-list-lverts* ((*v*,*e*)#*xs*) =
(*v* ≠ [] ∧ (∀ *v2* ∈ *fst* ‘ *set xs*. *set v* ∩ *set v2* = {}) ∧ *wf-list-lverts xs*)

## 8.2 List Dtrees as Well-Formed Dtrees

**lemma** *list-in-verts-if-lverts*: *x* ∈ *dlverts t* ⟹ (∃ *v* ∈ *dverts t*. *x* ∈ *set v*)
**by**(*induction t*) *fastforce*

**lemma** *list-in-verts-iff-lverts*: *x* ∈ *dlverts t* ⟷ (∃ *v* ∈ *dverts t*. *x* ∈ *set v*)
**by**(*induction t*) *fastforce*

**lemma** *lverts-if-in-verts*: ⟦*v* ∈ *dverts t*; *x* ∈ *set v*⟧ ⟹ *x* ∈ *dlverts t*
**by**(*induction t*) *fastforce*

**lemma** *nempty-inter-notin-dverts*: ⟦*v* ≠ []; *set v* ∩ *dlverts t* = {}⟧ ⟹ *v* ∉ *dverts t*
**using** *lverts-if-in-verts disjoint-iff-not-equal equals0I set-empty* **by** *metis*

**lemma** *empty-notin-wf-dlverts*: *wf-dlverts t* ⟹ [] ∉ *dverts t*
**by**(*induction t*) *auto*

**lemma** *wf-dlverts'-rec*: ⟦*wf-dlverts'* (*Node r xs*); *t1* ∈ *fst* ‘ *fset xs*⟧ ⟹ *wf-dlverts' t1*
**unfolding** *wf-dlverts'-def* **using** *wf-dverts-rec*[*of r xs t1*] *dverts-child-subseteq*[*of t1 xs*] **by** *blast*

**lemma** *wf-dlverts'-suc*: ⟦*wf-dlverts' t*; *t1* ∈ *fst* ‘ *fset* (*sucs t*)⟧ ⟹ *wf-dlverts' t1*
**using** *wf-dlverts'-rec*[*of root t sucs t*] **by** *simp*

**lemma** *wf-dlverts-suc*: ⟦*wf-dlverts t*; *t1* ∈ *fst* ‘ *fset* (*sucs t*)⟧ ⟹ *wf-dlverts t1*
**using** *wf-dlverts.simps*[*of root t sucs t*] **by** *auto*

**lemma** *wf-dlverts-subtree*: ⟦*wf-dlverts t*; *is-subtree t1 t*⟧ ⟹ *wf-dlverts t1*
**by** (*induction t*) *auto*

**lemma** *dlverts-eq-dverts-union*: *dlverts t* = ⋃ (*set ' dverts t*)
**by** (*induction t*) *fastforce*

**lemma** *dlverts-eq-dverts-union'*: *dlverts t* = (⋃*x*∈ *dverts t*. *set x*)
**using** *dlverts-eq-dverts-union* **by** *simp*

**lemma** *dverts-nempty*: *dverts t* ≠ {}
**using** *dtree.set*(*1*)[*of root t sucs t*] **by** *simp*

**lemma** *dlverts-nempty-aux*: [] ∉ *dverts t* ⟹ *dlverts t* ≠ {}
**using** *dverts-nempty dlverts-eq-dverts-union*[*of t*] **by** *fastforce*

**lemma** *dlverts-nempty-if-wf*: *wf-dlverts t* ⟹ *dlverts t* ≠ {}
**using** *dlverts-nempty-aux empty-notin-wf-dlverts* **by** *blast*

**lemma** *nempty-root-in-lverts*: *root t* ≠ [] ⟹ *hd* (*root t*) ∈ *dlverts t*
**using** *dtree.set-sel*(*1*) *list-in-verts-iff-lverts* **by** *fastforce*

**lemma** *roothd-in-lverts-if-wf*: *wf-dlverts t* ⟹ *hd* (*root t*) ∈ *dlverts t*
**using** *wf-dlverts.simps*[*of root t sucs t*] *nempty-root-in-lverts* **by** *auto*

**lemma** *hd-in-lverts-if-wf*: ⟦*wf-dlverts t*; *v* ∈ *dverts t*⟧ ⟹ *hd v* ∈ *dlverts t*
**using** *empty-notin-wf-dlverts hd-in-set*[*of v*] *lverts-if-in-verts* **by** *fast*

**lemma** *dlverts-notin-root-sucs*:
⟦*wf-dlverts t*; *t1* ∈ *fst ' fset* (*sucs t*); *x* ∈ *dlverts t1*⟧ ⟹ *x* ∉ *set* (*root t*)
**using** *wf-dlverts.simps*[*of root t sucs t*] **by** *fastforce*

**lemma** *dverts-inter-empty-if-verts-inter*:
**assumes** *dlverts x* ∩ *dlverts y* = {} **and** *wf-dlverts x*
**shows** *dverts x* ∩ *dverts y* = {}
**proof** (*rule ccontr*)
**assume** *asm*: *dverts x* ∩ *dverts y* ≠ {}
**then obtain** *r* **where** *r-def*: *r* ∈ *dverts x r* ∈ *dverts y* **by** *blast*
**then have** *r* ≠ [] **using** *assms*(*2*) **by**(*auto simp*: *empty-notin-wf-dlverts*)
**then obtain** *v* **where** *v-def*: *v* ∈ *set r* **by** *fastforce*
**then show** *False* **using** *r-def assms*(*1*) *lverts-if-in-verts* **by** (*metis IntI all-not-in-conv*)
**qed**

**lemma** *disjoint-dlverts-if-wf*: *wf-dlverts t* ⟹ *disjoint-dlverts* (*sucs t*)
**using** *wf-dlverts.simps*[*of root t sucs t*] **by** *simp*

**lemma** *disjoint-dlverts-subset*:
**assumes** *xs* |⊆| *ys* **and** *disjoint-dlverts ys*
**shows** *disjoint-dlverts xs*
**proof** (*rule ccontr*)
**assume** ¬ *disjoint-dlverts xs*
**then obtain** *x e1 y e2* **where** *x-def*: (*x*,*e1*) ∈ *fset xs* (*y*,*e2*) ∈ *fset xs*

*dlverts x* ∩ *dlverts y* ≠ {} ∧ (*x*,*e1*)≠(*y*,*e2*)
**by** *blast*
**have** (*x*,*e1*) ∈ *fset ys* (*y*,*e2*) ∈ *fset ys* **using** *x-def*(*1*,*2*) *assms*(*1*) *less-eq-fset.rep-eq* **by** *fast+*
**then show** *False* **using** *assms*(*2*) *x-def*(*3*) **by** *fast*
**qed**

**lemma** *root-empty-inter-subset*:
**assumes** *xs* |⊆| *ys* **and** ∀(*x*,*e1*) ∈ *fset ys*. *set r* ∩ *dlverts x* = {}
**shows** ∀(*x*,*e1*) ∈ *fset xs*. *set r* ∩ *dlverts x* = {}
**using** *assms less-eq-fset.rep-eq* **by** *force*

**lemma** *wf-dlverts-sub*:
**assumes** *xs* |⊆| *ys* **and** *wf-dlverts* (*Node r ys*)
**shows** *wf-dlverts* (*Node r xs*)
**proof** (*rule ccontr*)
**assume** *asm*: ¬*wf-dlverts* (*Node r xs*)
**have** *disjoint-dlverts xs* **using** *assms*(*2*) *disjoint-dlverts-subset*[*OF assms*(*1*)] **by** *simp*
**moreover have** *r* ≠ [] **using** *assms*(*2*) **by** *simp*
**moreover have** (∀(*x*,*e1*) ∈ *fset xs*. *set r* ∩ *dlverts x* = {})
**using** *assms*(*2*) *root-empty-inter-subset*[*OF assms*(*1*)] **by** *fastforce*
**ultimately obtain** *x e* **where** *x-def*: (*x*,*e*) ∈ *fset xs* ¬*wf-dlverts x* **using** *asm* **by** *auto*
**then have** (*x*,*e*) ∈ *fset ys* **using** *assms*(*1*) *fin-mono* **by** *metis*
**then show** *False* **using** *assms*(*2*) *x-def*(*2*) **by** *fastforce*
**qed**

**lemma** *wf-dlverts-sucs*: ⟦*wf-dlverts t*; *x* ∈ *fset* (*sucs t*)⟧ ⟹ *wf-dlverts* (*Node* (*root t*) {|*x*|})
**using** *wf-dlverts-sub*[*of* {|*x*|} *sucs t root t*] **by** (*simp add*: *less-eq-fset.rep-eq*)

**lemma** *wf-dverts-if-wf-dlverts*: *wf-dlverts t* ⟹ *wf-dverts t*
**proof**(*induction t*)
**case** (*Node r xs*)
**then have** ∀(*x*,*e*) ∈ *fset xs*. *wf-dverts x* **by** *auto*
**moreover have** ∀(*x*,*e*) ∈ *fset xs*. *r* ∉ *dverts x*
**using** *nempty-inter-notin-dverts Node.prems* **by** *fastforce*
**ultimately show** *?case*
**using** *Node.prems dverts-inter-empty-if-verts-inter wf-dverts-iff-dverts'*
**by** (*smt* (*verit*, *del-insts*) *wf-dlverts.simps wf-dverts'.simps case-prodD case-prodI2*)
**qed**

**lemma** *notin-dlverts-child-if-wf-in-root*:
⟦*wf-dlverts* (*Node r xs*); *x* ∈ *set r*; *t* ∈ *fst* ‘ *fset xs*⟧ ⟹ *x* ∉ *dlverts t*
**by** *fastforce*

**lemma** *notin-dlverts-suc-if-wf-in-root*:
⟦*wf-dlverts t1*; *x* ∈ *set* (*root t1*); *t2* ∈ *fst* ‘ *fset* (*sucs t1*)⟧ ⟹ *x* ∉ *dlverts t2*

**using** *notin-dlverts-child-if-wf-in-root*[*of root t1 sucs t1*] **by** *simp*

**lemma** *root-if-same-lvert-wf*:
⟦*wf-dlverts* (*Node r xs*); *x* ∈ *set r*; *v* ∈ *dverts* (*Node r xs*); *x* ∈ *set v*⟧ ⟹ *v* = *r*
**by** (*fastforce simp*: *lverts-if-in-verts dverts-child-if-not-root notin-dlverts-child-if-wf-in-root*)

**lemma** *dverts-same-if-set-wf*:
⟦*wf-dlverts t*; *v1* ∈ *dverts t*; *v2* ∈ *dverts t*; *x* ∈ *set v1*; *x* ∈ *set v2*⟧ ⟹ *v1* = *v2*
**proof**(*induction t*)
**case** (*Node r xs*)
**then show** *?case*
**proof**(*cases x* ∈ *set r*)
**case** *True*
**then show** *?thesis* **using** *Node.prems*(*2*,*3*,*4*,*5*) *root-if-same-lvert-wf*[*OF Node.prems*(*1*)]
**by** *blast*
**next**
**case** *False*
**then obtain** *t2 e2* **where** *t2-def*: (*t2*,*e2*) ∈ *fset xs x* ∈ *dlverts t2*
**using** *Node.prems*(*2*,*4*) *lverts-if-in-verts* **by** *fastforce*
**then have** ∀ (*t3*,*e3*)∈*fset xs*. (*t3*,*e3*) = (*t2*,*e2*) ∨ *x* ∉ *dlverts t3*
**using** *Node.prems*(*1*) **by** *fastforce*
**then have** *v1* ∈ *dverts t2* ∧ *v2* ∈ *dverts t2*
**using** *Node.prems*(*2*−*5*) *lverts-if-in-verts False* **by** *force*
**then show** *?thesis* **using** *Node.IH t2-def*(*1*) *Node.prems*(*1*,*4*,*5*) **by** *auto*
**qed**
**qed**

**lemma** *dtree-from-list-empty-inter-iff*:
(∀ *v* ∈ *fst* ‘ *set* ((*v*, *e*) # *xs*). *set r* ∩ *set v* = {})
⟷ (∀ (*x*,*e1*) ∈ *fset* {|(*dtree-from-list v xs*,*e*)|}. *set r* ∩ *dlverts x* = {}) (**is** *?P* ⟷ *?Q*)
**proof**
**assume** *asm*: *?P*
**have** *dverts* (*dtree-from-list v xs*) = *fst* ‘ *set* ((*v*,*e*)#*xs*)
**by**(*simp add*: *dtree-from-list-eq-dverts*)
**then show** *?Q* **using** *list-in-verts-if-lverts asm* **by** *fastforce*
**next**
**assume** *asm*: *?Q*
**have** *dverts* (*dtree-from-list v xs*) = *fst* ‘ *set* ((*v*,*e*)#*xs*)
**by**(*simp add*: *dtree-from-list-eq-dverts*)
**moreover have** (*dtree-from-list v xs*,*e*) ∈ *fset* {|(*dtree-from-list v xs*, *e*)|} **by** *simp*
**ultimately show** *?P* **using** *asm lverts-if-in-verts* **by** *fast*
**qed**

**lemma** *wf-dlverts-iff-wf-list-lverts*:
(∀ *v* ∈ *fst* ‘ *set xs*. *set r* ∩ *set v* = {}) ∧ *r* ≠ [] ∧ *wf-list-lverts xs*
⟷ *wf-dlverts* (*dtree-from-list r xs*)
**proof**(*induction xs arbitrary*: *r rule*: *wf-list-lverts.induct*)

**case** (*2 v e xs*)
**then show** *?case* **using** *dtree-from-list-empty-inter-iff*[*of v e*] **by** *auto*
**qed** (*simp*)

**lemma** *vert-disjoint-if-not-root*:
**assumes** *wf-dlverts t*
**and** $v \in$ *dverts t* $-$ {*root t*}
**shows** *set* (*root t*) $\cap$ *set v* = {}
**proof** −
**obtain** *t1 e1* **where** *t1-def*: (*t1*,*e1*) $\in$ *fset* (*sucs t*) *v* $\in$ *dverts t1*
**using** *assms*(*2*) *dtree.set-cases*(*1*) **by** *force*
**then show** *?thesis* **using** *assms*(*1*) *wf-dlverts.simps*[*of root t*] *lverts-if-in-verts*
**by** *fastforce*
**qed**

**lemma** *vert-disjoint-if-to-list*:
⟦*wf-dlverts* (*Node r* {|(*t1*,*e1*)|}); *v* $\in$ *fst* ‘ *set* (*dtree-to-list t1*)⟧
$\Longrightarrow$ *set* (*root t1*) $\cap$ *set v* = {}
**using** *vert-disjoint-if-not-root dtree-to-list-sub-dverts wf-dverts-if-wf-dlverts* **by**
*fastforce*

**lemma** *wf-list-lverts-if-wf-dlverts*: *wf-dlverts t* $\Longrightarrow$ *wf-list-lverts* (*dtree-to-list t*)
**proof**(*induction t*)
**case** (*Node r xs*)
**then show** *?case*
**proof**(*cases* $\forall$ *x*. *xs* $\neq$ {|*x*|})
**case** *True*
**then show** *?thesis* **using** *dtree-to-list.simps*(*2*) **by** *simp*
**next**
**case** *False*
**then obtain** *t1 e1* **where** *t1-def*: *xs* = {|(*t1*,*e1*)|} **by** *auto*
**then have** *wf-dlverts t1* **using** *Node.prems* **by** *simp*
**then have** *root t1* $\neq$ [] **using** *wf-dlverts.simps*[*of root t1 sucs t1*] **by** *simp*
**then show** *?thesis* **using** *Node vert-disjoint-if-to-list t1-def* **by** *fastforce*
**qed**
**qed**

**lemma** *child-in-dlverts*: (*t1*,*e*) $\in$ *fset xs* $\Longrightarrow$ *dlverts t1* $\subseteq$ *dlverts* (*Node r xs*)
**by** *force*

**lemma** *suc-in-dlverts*: (*t1*,*e*) $\in$ *fset* (*sucs t2*) $\Longrightarrow$ *dlverts t1* $\subseteq$ *dlverts t2*
**using** *child-in-dlverts*[*of t1 e sucs t2 root t2*] **by** *auto*

**lemma** *suc-in-dlverts′*: *t1* $\in$ *fst* ‘ *fset* (*sucs t2*) $\Longrightarrow$ *dlverts t1* $\subseteq$ *dlverts t2*
**using** *suc-in-dlverts* **by** *fastforce*

**lemma** *subtree-in-dlverts*: *is-subtree t1 t2* $\Longrightarrow$ *dlverts t1* $\subseteq$ *dlverts t2*
**by**(*induction t2*) *fastforce*

**lemma** *subtree-root-if-dlverts*: $x \in dlverts\ t \Longrightarrow \exists\, r\ xs.\ is\text{-}subtree\ (Node\ r\ xs)\ t \wedge x \in set\ r$
**using** *subtree-root-if-dverts list-in-verts-if-lverts* **by** *fast*

**lemma** *x-not-root-strict-subtree*:
**assumes** $x \in dlverts\ t$ **and** $x \notin set\ (root\ t)$
**shows** $\exists\, r\ xs\ t1.\ is\text{-}subtree\ (Node\ r\ xs)\ t \wedge t1 \in fst\ `\ fset\ xs \wedge x \in set\ (root\ t1)$
**proof** −
**obtain** *r xs* **where** *r-def*: *is-subtree* (*Node r xs*) *t x* ∈ *set r*
**using** *subtree-root-if-dlverts*[*OF assms*(*1*)] **by** *fast*
**then have** *sub*: *strict-subtree* (*Node r xs*) *t* **using** *assms*(*2*) *strict-subtree-def* **by** *fastforce*
**then show** *?thesis* **using** *assms*(*2*) *subtree-child-if-strict-subtree*[*OF sub*] *r-def*(*2*) **by** *force*
**qed**

**lemma** *dverts-disj-if-wf-dlverts*:
⟦*wf-dlverts t*; *v1* ∈ *dverts t*; *v2* ∈ *dverts t*; *v1* ≠ *v2*⟧ ⟹ *set v1* ∩ *set v2* = {}
**using** *dverts-same-if-set-wf* **by** *fast*

**thm** *empty-notin-wf-dlverts*

**lemma** *wf-dlverts'-if-dlverts*: *wf-dlverts t* ⟹ *wf-dlverts' t*
**using** *wf-dlverts'-def empty-notin-wf-dlverts dverts-disj-if-wf-dlverts wf-dverts-if-wf-dlverts*
**by** *blast*

**lemma** *disjoint-dlverts-if-wf'-aux*:
**assumes** *wf-dlverts'* (*Node r xs*)
**and** (*t1*,*e1*) ∈ *fset xs*
**and** (*t2*,*e2*) ∈ *fset xs*
**and** (*t1*,*e1*) ≠ (*t2*,*e2*)
**shows** *dlverts t1* ∩ *dlverts t2* = {}
**proof**(*rule ccontr*)
**assume** *dlverts t1* ∩ *dlverts t2* ≠ {}
**then obtain** *x y* **where** *x-def*: *x* ∈ *dverts t1 y* ∈ *dverts t2 set x* ∩ *set y* ≠ {}
**using** *dlverts-eq-dverts-union*[*of t1*] *dlverts-eq-dverts-union*[*of t2*] **by** *auto*
**then have** *x* ∈ *dverts* (*Node r xs*) *y* ∈ *dverts* (*Node r xs*)
**using** *dverts-child-subseteq assms*(*2*,*3*) **by** *auto*
**moreover have** *x* ≠ *y*
**using** *assms*(*1*) *disjoint-dverts-if-wf-aux*[*rotated*, *OF assms*(*2*−*4*)] *x-def*(*1*,*2*)
**unfolding** *wf-dlverts'-def* **by** *blast*
**ultimately show** *False* **using** *assms*(*1*) *x-def*(*3*) **unfolding** *wf-dlverts'-def* **by** *blast*
**qed**

**lemma** *disjoint-dlverts-if-wf'*: *wf-dlverts'* (*Node r xs*) ⟹ *disjoint-dlverts xs*
**using** *disjoint-dlverts-if-wf'-aux* **by** *fast*

**lemma** *root-nempty-if-wf'*: *wf-dlverts'* (*Node r xs*) ⟹ *r* ≠ []

**unfolding** *wf-dlverts'-def* **by** *fastforce*

**lemma** *disjoint-root-if-wf'-aux*:
**assumes** *wf-dlverts'* (*Node r xs*)
**and** (*t1*,*e1*) ∈ *fset xs*
**shows** *set r* ∩ *dlverts t1* = {}
**proof**(*rule ccontr*)
**assume** *set r* ∩ *dlverts t1* ≠ {}
**then obtain** *x* **where** *x-def*: *x* ∈ *dverts t1 set x* ∩ *set r* ≠ {}
**using** *dlverts-eq-dverts-union* **by** *fast*
**then have** *x* ∈ *dverts* (*Node r xs*) **using** *dverts-child-subseteq assms*(*2*) **by** *auto*
**moreover have** *r* ∈ *dverts* (*Node r xs*) **by** *simp*
**moreover have** *x* ≠ *r*
**using** *assms x-def*(*1*) *root-not-child-if-wf-dverts* **unfolding** *wf-dlverts'-def* **by** *fast*
**ultimately show** *False* **using** *assms*(*1*) *x-def*(*2*) **unfolding** *wf-dlverts'-def* **by** *blast*
**qed**

**lemma** *disjoint-root-if-wf'*:
*wf-dlverts'* (*Node r xs*) ⟹ ∀ (*t1*,*e1*) ∈ *fset xs*. *set r* ∩ *dlverts t1* = {}
**using** *disjoint-root-if-wf'-aux* **by** *fast*

**lemma** *wf-dlverts-if-dlverts'*: *wf-dlverts' t* ⟹ *wf-dlverts t*
**proof**(*induction t*)
**case** (*Node r xs*)
**then have** ∀ (*t1*,*e1*) ∈ *fset xs*. *set r* ∩ *dlverts t1* = {}
**using** *disjoint-root-if-wf'* **by** *blast*
**moreover have** *r* ≠ [] ∧ *disjoint-dlverts xs*
**using** *disjoint-dlverts-if-wf' Node.prems root-nempty-if-wf'* **by** *fast*
**moreover have** ∀ (*t1*,*e1*) ∈ *fset xs*. *wf-dlverts t1*
**using** *Node wf-dlverts'-rec* **by** *fastforce*
**ultimately show** *?case* **by** *auto*
**qed**

**lemma** *wf-dlverts-iff-dlverts'*: *wf-dlverts t* ⟷ *wf-dlverts' t*
**using** *wf-dlverts-if-dlverts' wf-dlverts'-if-dlverts* **by** *blast*

**locale** *list-dtree* =
**fixes** *t* :: ('*a list*,'*b*) *dtree*
**assumes** *wf-arcs*: *wf-darcs t*
**and** *wf-lverts*: *wf-dlverts t*

**sublocale** *list-dtree* ⊆ *wf-dtree*
**using** *wf-arcs wf-lverts wf-dverts-if-wf-dlverts* **by**(*unfold-locales*) *auto*

**theorem** *list-dtree-iff-wf-list*:
*wf-list-arcs xs* ∧ (∀ *v* ∈ *fst* ' *set xs*. *set r* ∩ *set v* = {}) ∧ *r* ≠ [] ∧ *wf-list-lverts xs*
⟷ *list-dtree* (*dtree-from-list r xs*)

**using** *wf-darcs-iff-wf-list-arcs wf-dlverts-iff-wf-list-lverts list-dtree-def* **by** *metis*

**lemma** *list-dtree-subset*:
**assumes** *xs* $|\subseteq|$ *ys* **and** *list-dtree* (*Node r ys*)
**shows** *list-dtree* (*Node r xs*)
**using** *wf-dlverts-sub*[*OF assms*(*1*)] *wf-darcs-sub*[*OF assms*(*1*)] *assms*(*2*)
**by** (*unfold-locales*) (*fast dest*: *list-dtree.wf-lverts list-dtree.wf-arcs*)+

**context** *fin-list-directed-tree*
**begin**

**lemma** *dlverts-disjoint*:
**assumes** $r \in$ *verts T* **and** (*Node r xs*) = *to-dtree-aux r*
**and** (*x*,*e1*) ∈ *fset xs* **and** (*y*,*e2*) ∈ *fset xs* **and** (*x*,*e1*)≠(*y*,*e2*)
**shows** *dlverts x* ∩ *dlverts y* = {}
**proof** (*rule ccontr*)
**assume** *dlverts x* ∩ *dlverts y* ≠ {}
**then obtain** *v* **where** *v-def*[*simp*]: *v* ∈ *dlverts x v* ∈ *dlverts y* **by** *blast*
**obtain** *x1* **where** *x1-def*: *v* ∈ *set x1 x1* ∈ *dverts x* **using** *list-in-verts-if-lverts*
**by** *force*
**obtain** *y1* **where** *y1-def*: *v* ∈ *set y1 y1* ∈ *dverts y* **using** *list-in-verts-if-lverts*
**by** *force*
**have** *0*: *y* = *to-dtree-aux* (*Dtree.root y*) **using** *to-dtree-aux-self assms*(*2*,*4*) **by**
*blast*
**have** *r* $\rightarrow_T$ *Dtree.root y*
**using** *assms*(*2*,*4*) *dominated-if-child* **by** (*metis* (*no-types*, *opaque-lifting*) *fst-conv*
*image-iff*)
**then have** *1*: *Dtree.root y* ∈ *verts T* **using** *adj-in-verts*(*2*) **by** *simp*
**have** *r* $\rightarrow_T$ *Dtree.root x*
**using** *assms*(*2*,*3*) *dominated-if-child* **by** (*metis* (*no-types*, *opaque-lifting*) *fst-conv*
*image-iff*)
**then have** *Dtree.root x* ∈ *verts T* **using** *adj-in-verts*(*2*) **by** *simp*
**moreover have** *x* = *to-dtree-aux* (*Dtree.root x*) **using** *to-dtree-aux-self assms*(*2*,*3*)
**by** *blast*
**ultimately have** *Dtree.root x* $\rightarrow^*{}_T$ *x1* **using** *to-dtree-aux-dverts-reachable x1-def*(*2*)
**by** *blast*
**moreover have** *Dtree.root y* $\rightarrow^*{}_T$ *y1* **using** *0 1 to-dtree-aux-dverts-reachable*
*y1-def*(*2*) **by** *blast*
**ultimately have** *x1* = *y1* **using** *disjoint-verts reachable-in-verts*(*2*) *x1-def*(*1*)
*y1-def*(*1*) **by** *auto*
**then show** *False* **using** *dverts-disjoint*[*OF assms*(*2*−*5*)] *x1-def*(*2*) *y1-def*(*2*) **by**
*blast*
**qed**

**lemma** *wf-dlverts-to-dtree-aux*: ⟦*r* ∈ *verts T*; *t* = *to-dtree-aux r*⟧ ⟹ *wf-dlverts t*
**proof**(*induction t arbitrary*: *r rule*: *darcs-mset.induct*)
**case** (*1 r′ xs*)
**then have** *r* = *r′* **by** *simp*
**have** ∀(*x*,*e*) ∈ *fset xs*. *wf-dlverts x* ∧ *set r* ∩ *dlverts x* = {}

**proof** (*standard*, *standard*, *standard*)
**fix** *xp x e*
**assume** *asm*: *xp* ∈ *fset xs xp* = (*x*,*e*)
**then have** *0*: *x* = *to-dtree-aux* (*Dtree.root x*) **using** *to-dtree-aux-self 1.prems*(*2*)
**by** *simp*
**have** *2*: *r* →$_T$ *Dtree.root x* **using** *asm 1.prems* ‹*r* = *r*′›
**by** (*metis* (*no-types*, *opaque-lifting*) *dominated-if-child fst-conv image-iff*)
**then have** *3*: *Dtree.root x* ∈ *verts T* **using** *adj-in-verts*(*2*) **by** *simp*
**then show** *wf-dlverts x* **using** *1.IH asm 0* **by** *blast*
**have** *r* ∉ *dverts x*
**proof**
**assume** *r* ∈ *dverts x*
**then have** *Dtree.root x* →*$^+_T$ r* **using** *0 3 to-dtree-aux-dverts-reachable* **by**
*blast*
**then have** *r* →*$^+_T$ r* **using** *2* **by** *auto*
**then show** *False* **using** *reachable1-not-reverse* **by** *blast*
**qed**
**then show** *set r* ∩ *dlverts x* = {}
**using** *0 1.prems*(*1*) *3 disjoint-iff-not-equal disjoint-verts list-in-verts-if-lverts*
**by** (*metis reachable-in-verts*(*2*) *to-dtree-aux-dverts-reachable*)
**qed**
**moreover have** *disjoint-dlverts xs* **using** *dlverts-disjoint 1.prems* **by** *fastforce*
**ultimately show** *?case* **using** ‹*r* = *r*′› **by** (*auto simp add*: *1.prems*(*1*) *nempty-verts*)
**qed**

**lemma** *wf-dlverts-to-dtree*: *wf-dlverts to-dtree*
**using** *to-dtree-def wf-dlverts-to-dtree-aux root-in-T* **by** *blast*

**theorem** *list-dtree-to-dtree*: *list-dtree to-dtree*
**using** *list-dtree-def wf-dlverts-to-dtree wf-darcs-to-dtree* **by** *blast*

**end**

**context** *list-dtree*
**begin**

**lemma** *list-dtree-rec*: ⟦*Node r xs* = *t*; (*x*,*e*) ∈ *fset xs*⟧ ⟹ *list-dtree x*
**using** *wf-arcs wf-lverts* **by**(*unfold-locales*) *auto*

**lemma** *list-dtree-rec-suc*: (*x*,*e*) ∈ *fset* (*sucs t*) ⟹ *list-dtree x*
**using** *list-dtree-rec*[*of root t*] **by** *force*

**lemma** *list-dtree-sub*: *is-subtree x t* ⟹ *list-dtree x*
**using** *list-dtree-axioms* **proof**(*induction t rule*: *darcs-mset.induct*)
**case** (*1 r xs*)
**then interpret** *list-dtree Node r xs* **by** *blast*
**show** *?case*
**proof**(*cases x* = *Node r xs*)
**case** *True*

**then show** *?thesis* **by** (*simp add*: *1.prems*)
**next**
**case** *False*
**then show** *?thesis* **using** *1.IH list-dtree-rec 1.prems(1)* **by** *auto*
**qed**
**qed**

**theorem** *from-dtree-fin-list-dir*: *fin-list-directed-tree* (*root t*) (*from-dtree dt dh t*)
**unfolding** *fin-list-directed-tree-def fin-list-directed-tree-axioms-def*
**by** (*auto simp*: *from-dtree-fin-directed empty-notin-wf-dlverts*[*OF wf-lverts*]
*intro*: *wf-lverts dverts-same-if-set-wf*)

## 8.3 Combining Preserves Well-Formedness

**lemma** *remove-child-sub*: *remove-child x xs* |⊆| *xs*
**by** *auto*

**lemma** *child2-commute-aux*:
**assumes** $f = (\lambda(t,\text{-})\ b.$ *case t of Node r ys* ⇒ *if r = a then ys* |∪| *b else b*)
**shows** $(f\ y \circ f\ x)\ z = (f\ x \circ f\ y)\ z$
**proof** −
**obtain** *r1 ys1 e1* **where** *y-def*: $y = (Node\ r1\ ys1,\ e1)$ **by** (*metis dtree.exhaust eq-snd-iff*)
**obtain** *r2 ys2 e2* **where** $x = (Node\ r2\ ys2,\ e2)$ **by** (*metis dtree.exhaust eq-snd-iff*)
**then show** *?thesis* **by** (*simp add*: *assms funion-left-commute y-def*)
**qed**

**lemma** *child2-commute*:
*comp-fun-commute* $(\lambda(t,\text{-})\ b.$ *case t of Node r ys* ⇒ *if r = x then ys* |∪| *b else b*)
**using** *comp-fun-commute-def child2-commute-aux* **by** *fastforce*

**interpretation** *Comm*:
*comp-fun-commute* $\lambda(t,\text{-})\ b.$ *case t of Node r ys* ⇒ *if r = x then ys* |∪| *b else b*
**by** (*rule child2-commute*)

**lemma** *input-in-child2*:
*zs* |⊆| *child2 x zs ys*
**proof**(*induction ys*)
**case** *empty*
**then show** *?case* **using** *Comm.ffold-empty* **by** *simp*
**next**
**case** (*insert y ys*)
**then obtain** *r xs e* **where** *r-def*: $(Node\ r\ xs,e) = y$ **by** (*metis dtree.exhaust surj-pair*)
**let** *?f* = $(\lambda(t,\text{-})\ b.$ *case t of Node r ys* ⇒ *if r = x then ys* |∪| *b else b*)
**show** *?case*
**proof**(*cases r=x*)
**case** *True*
**then have** *ffold ?f zs* (*finsert y ys*) = *xs* |∪| (*ffold ?f zs ys*)

```
    using r-def insert.hyps by force
   then show ?thesis using insert.IH by blast
  next
   case False
   then have ffold ?f zs (finsert y ys) = (ffold ?f zs ys) using r-def insert.hyps
by force
   then show ?thesis using insert.IH by blast
  qed
qed

lemma child2-subset-if-input1:
  zs' |⊆| zs ⟹ child2 x zs' ys |⊆| child2 x zs ys
proof(induction ys)
  case (insert y ys)
  obtain r xs e where r-def: (Node r xs, e) = y by (metis dtree.exhaust surj-pair)
  let ?f = (λ(t,-) b. case t of Node r ys ⇒ if r = x then ys |∪| b else b)
  show ?case
  proof(cases r=x)
   case True
   then have ffold ?f zs (finsert y ys) = xs |∪| (ffold ?f zs ys)
    using r-def insert.hyps by force
   moreover have ffold ?f zs' (finsert y ys) = xs |∪| (ffold ?f zs' ys)
    using r-def insert.hyps True by force
   ultimately show ?thesis using insert by blast
  next
   case False
   then have ffold ?f zs (finsert y ys) = (ffold ?f zs ys) using r-def insert.hyps
by force
   moreover have ffold ?f zs' (finsert y ys) = (ffold ?f zs' ys)
    using r-def insert.hyps False by force
   ultimately show ?thesis using insert by blast
  qed
qed (simp)

lemma child2-subset-if-input2:
  ys' |⊆| ys ⟹ child2 x xs ys' |⊆| child2 x xs ys
proof(induction fcard ys arbitrary: ys)
  case (Suc n)
  show ?case
  proof(cases ys' = ys)
   case False
   then obtain z where z-def: z |∈| ys ∧ z |∉| ys' using Suc.prems by blast
   then obtain zs where zs-def: finsert z zs = ys ∧ z |∉| zs by blast
   then have ys' |⊆| zs ∧ fcard zs = n
    using Suc.prems(1) Suc.hyps(2) z-def fcard-finsert-disjoint by fastforce
   then have 0: child2 x xs ys' |⊆| child2 x xs zs using Suc.hyps(1) by blast
  obtain r rs e where r-def: (Node r rs, e) = z by (metis dtree.exhaust surj-pair)
   then show ?thesis using 0 zs-def by force
  qed (simp)
```

**qed** (*simp*)

**lemma** *darcs-split*: *darcs* (*Node r* (*xs*|∪|*ys*)) = *darcs* (*Node r xs*) ∪ *darcs* (*Node r ys*)
**by** *simp*

**lemma** *darcs-sub-if-children-sub*: *xs* |⊆| *ys* ⟹ *darcs* (*Node r xs*) ⊆ *darcs* (*Node v ys*)
**proof**(*induction fcard ys arbitrary*: *ys*)
**case** (*Suc n*)
**then show** *?case*
**proof**(*cases ys* = *xs*)
**case** *False*
**then obtain** *z* **where** *z-def*: *z* |∈| *ys* ∧ *z* |∉| *xs* **using** *Suc.prems* **by** *blast*
**then obtain** *zs* **where** *zs-def*: *finsert z zs* = *ys* ∧ *z* |∉| *zs* **by** *blast*
**then have** *xs* |⊆| *zs* ∧ *fcard zs* = *n*
**using** *Suc.prems*(*1*) *Suc.hyps*(*2*) *z-def fcard-finsert-disjoint* **by** *fastforce*
**then have** *darcs* (*Node r xs*) ⊆ *darcs* (*Node v zs*) **using** *Suc.hyps*(*1*) **by** *blast*
**then show** *?thesis* **using** *zs-def darcs-split*[*of v* {|*z*|} *zs*] **by** *auto*
**qed** (*simp*)
**qed** (*simp*)

**lemma** *darc-in-child2-snd-if-nin-fst*:
*e* ∈ *darcs* (*Node x* (*child2 a xs ys*)) ⟹ *e* ∉ *darcs* (*Node v ys*) ⟹ *e* ∈ *darcs* (*Node r xs*)
**proof**(*induction ys*)
**case** (*insert y ys*)
**obtain** *r rs e1* **where** *r-def*: (*Node r rs*, *e1*) = *y* **by** (*metis dtree.exhaust surj-pair*)
**then have** *e-not-rs*: *e* ∉ *darcs* (*Node x rs*) **using** *insert.prems*(*2*) **by** *fastforce*
**show** *?case*
**proof**(*cases r* = *a*)
**case** *True*
**then have** *darcs* (*Node x* (*child2 a xs* (*finsert y ys*)))
= *darcs* (*Node x* (*rs* |∪| (*child2 a xs ys*)))
**using** *r-def insert.hyps*(*1*) **by** *force*
**moreover have** ... = *darcs* (*Node x rs*) ∪ *darcs* (*Node x* (*child2 a xs ys*)) **by** *simp*
**ultimately have** *e* ∈ *darcs* (*Node x* (*child2 a xs ys*)) **using** *insert.prems*(*1*) *e-not-rs* **by** *blast*
**then show** *?thesis* **using** *insert.IH insert.prems*(*2*) **by** *simp*
**next**
**case** *False*
**then have** *darcs* (*Node x* (*child2 a xs* (*finsert y ys*))) = *darcs* (*Node x* (*child2 a xs ys*))
**using** *r-def insert.hyps*(*1*) **by** *force*
**then show** *?thesis* **using** *insert.IH insert.prems* **by** *simp*
**qed**
**qed** (*simp*)

**lemma** *darc-in-child2-fst-if-nin-snd*:
  $e \in$ *darcs* (*Node x* (*child2 a xs ys*)) $\Longrightarrow$ $e \notin$ *darcs* (*Node v xs*) $\Longrightarrow$ $e \in$ *darcs* (*Node r ys*)
  **using** *darc-in-child2-snd-if-nin-fst* **by** *fast*

**lemma** *darcs-child2-sub*: *darcs* (*Node x* (*child2 y xs ys*)) $\subseteq$ *darcs* (*Node r xs*) $\cup$ *darcs* (*Node r' ys*)
  **using** *darc-in-child2-snd-if-nin-fst* **by** *fast*

**lemma** *darcs-combine-sub-orig*: *darcs* (*combine x y t1*) $\subseteq$ *darcs t1*
**proof**(*induction t1*)
  **case** *ind*: (*Node r xs*)
  **show** *?case*
  **proof**(*cases x=r* $\wedge$ ($\exists$ *t*. *t* $\in$ *fst* ‘ *fset xs* $\wedge$ *root t* = *y*))
    **case** *True*
    **then have** *darcs* (*combine x y* (*Node r xs*))
          = *darcs* (*Node* (*x@y*) (*child2 y* (*remove-child y xs*) *xs*)) **by** *simp*
    **also have** ... $\subseteq$ *darcs* (*Node x* (*child2 y xs xs*))
      **using** *darcs-sub-if-children-sub*[*of child2 y* (*remove-child y xs*) *xs child2 y xs xs*]
        *child2-subset-if-input1*[*of remove-child y xs xs*] *remove-child-sub* **by** *fast*
    **finally show** *?thesis* **using** *darcs-child2-sub* **by** *fast*
  **next**
    **case** *False*
    **then have** *darcs* (*combine x y* (*Node r xs*))
          = *darcs* (*Node r* ((*λ*(*t*,*e*). (*combine x y t*,*e*)) |‘| *xs*))
      **by** *auto*
    **also have** ... $\subseteq$ ($\bigcup$(*t*,*e*)$\in$*fset xs*. $\bigcup$ (*darcs* ‘ {*t*}) $\cup$ {*e*})
      **using** *ind.IH wf-dtree-rec* **by** *fastforce*
    **finally show** *?thesis* **by** *force*
  **qed**
**qed**

**lemma** *child2-in-child*:
  ⟦*b* $\in$ *fset* (*child2 a ys xs*); *b* |$\notin$| *ys*⟧ $\Longrightarrow$ $\exists$ *rs e*. (*Node a rs*, *e*) $\in$ *fset xs* $\wedge$ *b* |$\in$| *rs*
**proof**(*induction xs*)
  **case** (*insert x xs*)
   **obtain** *r rs e1* **where** *r-def*: (*Node r rs*, *e1*) = *x* **by** (*metis dtree.exhaust surj-pair*)
  **show** *?case*
  **proof**(*cases r* = *a*)
    **case** *ra*: *True*
    **then have** *0*: *child2 a ys* (*finsert x xs*) = *rs* |$\cup$| (*child2 a ys xs*)
      **using** *r-def insert.hyps*(*1*) **by** *force*
    **show** *?thesis*
    **proof**(*cases b* |$\in$| *rs*)
      **case** *True*
      **then show** *?thesis* **using** *r-def ra* **by** *auto*

```
    next
      case False
      then have b ∈ fset (child2 a ys xs) using insert.prems(1) 0 by force
      then show ?thesis using insert.IH insert.prems(2) by auto
    qed
  next
    case False
    then show ?thesis using insert r-def by force
  qed
qed simp

lemma child-in-darcs: (y,e2) ∈ fset xs ⟹ darcs y ∪ {e2} ⊆ darcs (Node r xs)
  by force

lemma disjoint-darcs-child2:
  assumes wf-darcs (Node r xs)
  shows disjoint-darcs (child2 a (remove-child a xs) xs) (is disjoint-darcs ?P)
proof (rule ccontr)
  assume ¬ disjoint-darcs ?P
  then obtain x e1 y e2 where asm: (x,e1) ∈ fset ?P (y,e2) ∈ fset ?P (e1 ∈
darcs x ∨
      ((darcs x ∪ {e1}) ∩ (darcs y ∪ {e2}) ≠ {} ∧ (x,e1)≠(y,e2))) by blast
  note wf-darcs-iff-darcs'[simp]
  consider (x,e1) ∈ fset (remove-child a xs) e1 ∈ darcs x
    | (x,e1) ∈ fset (remove-child a xs) e1 ∉ darcs x (y,e2) ∈ fset (remove-child a
xs)
    | (x,e1) ∈ fset (remove-child a xs) e1 ∉ darcs x (y,e2) |∉| (remove-child a xs)
    | (x,e1) |∉| (remove-child a xs) e1 ∈ darcs x
    | (x,e1) |∉| (remove-child a xs) e1 ∉ darcs x (y,e2) ∈ fset (remove-child a xs)
    | (x,e1) |∉| (remove-child a xs) e1 ∉ darcs x (y,e2) |∉| (remove-child a xs)
    by auto
  then show False
  proof(cases)
    case 1
    then show ?thesis using assms by auto
  next
    case 2
    then show ?thesis using assms asm(3) by fastforce
  next
    case 3
    then have x-xs: (x,e1) ∈ fset xs by simp
    obtain rs2 re2 where r2-def: (Node a rs2, re2) ∈ fset xs (y,e2) |∈| rs2
      using child2-in-child asm(2) 3(3) by fast
    then have darcs y ∪ {e2} ⊆ darcs (Node a rs2) using child-in-darcs by fast
    then have (darcs x ∪ {e1}) ∩ (darcs (Node a rs2) ∪ {re2}) ≠ {} using 3(2)
asm(3) by blast
    moreover have (x,e1)≠(Node a rs2, re2) using 3(1) by force
    ultimately have ¬ disjoint-darcs xs using r2-def(1) x-xs by fast
    then show ?thesis using assms by simp
```

```
  next
    case 4
    then obtain rs1 re1 where r1-def: (Node a rs1, re1) ∈ fset xs (x,e1) |∈| rs1
      using child2-in-child asm(1) by fast
    then have ¬disjoint-darcs rs1 using 4(2) by fast
    then show ?thesis using assms r1-def(1) by fastforce
  next
    case 5
    then obtain rs1 re1 where r1-def: (Node a rs1, re1) ∈ fset xs (x,e1) |∈| rs1
      using child2-in-child asm(1) by fast
    have 1: (darcs (Node a rs1) ∪ {re1}) ∩ (darcs y ∪ {e2}) ≠ {}
      using r1-def(2) asm(3) 5(2) child-in-darcs by fast
    have y-xs: (y,e2) ∈ fset xs using 5(3) by simp
    then have (Node a rs1, re1)≠(y,e2) using 5(3) by force
    then have ¬ disjoint-darcs xs using r1-def(1) y-xs 1 by fast
    then show ?thesis using assms by simp
  next
    case 6
    then obtain rs1 re1 where r1-def: (Node a rs1, re1) ∈ fset xs (x,e1) |∈| rs1
      using child2-in-child asm(1) by fast
    then have 1: (darcs (Node a rs1) ∪ {re1}) ∩ (darcs y ∪ {e2}) ≠ {}
      using asm(3) 6(2) child-in-darcs by fast
    obtain rs2 re2 where r2-def: (Node a rs2, re2) ∈ fset xs (y,e2) |∈| rs2
      using child2-in-child asm(2) 6(3) by fast
    then have darcs y ∪ {e2} ⊆ darcs (Node a rs2) using child-in-darcs by fast
    then have 1: (darcs (Node a rs1) ∪ {re1}) ∩ (darcs (Node a rs2) ∪ {re2}) ≠
{}
      using 1 asm(3) 6(2) child-in-darcs by blast
    then show ?thesis
    proof(cases (Node a rs1, re1) = (Node a rs2, re2))
      case True
      then have (x,e1) ∈ fset rs1 ∧ (y,e2) ∈ fset rs1
        using r1-def(2) r2-def(2) by fast
      then show ?thesis using assms r1-def asm(3) 6(2) by fastforce
    next
      case False
      then have ¬ disjoint-darcs xs using r1-def(1) r2-def(1) 1 by fast
      then show ?thesis using assms by simp
    qed
  qed
qed

lemma wf-darcs-child2:
  assumes wf-darcs (Node r xs) and (x,e) ∈ fset (child2 a (remove-child a xs) xs)
  shows wf-darcs x
proof(cases (x,e) |∈| remove-child a xs)
  case True
  then show ?thesis using assms(1) by (fastforce simp: wf-darcs-iff-darcs')
next
```

```
  case False
  then obtain r rs e1  where (Node r rs, e1) ∈ fset xs ∧ (x,e) |∈| rs ∧ r = a
    using child2-in-child assms(2) by fast
  then show ?thesis using assms by (fastforce simp: wf-darcs-iff-darcs')
qed

lemma disjoint-darcs-combine:
  assumes Node r xs = t
  shows disjoint-darcs ((λ(t,e). (combine x y t,e)) |`| xs)
proof −
  have disjoint-darcs xs using wf-arcs assms by (fastforce simp: wf-darcs-iff-darcs')
  then show ?thesis
   using disjoint-darcs-img[of xs combine x y] by (simp add: darcs-combine-sub-orig)
qed

lemma wf-darcs-combine: wf-darcs (combine x y t)
using list-dtree-axioms proof(induction t)
  case ind: (Node r xs)
  then interpret list-dtree Node r xs using ind.prems by blast
  show ?case
  proof(cases x=r ∧ (∃ t. t ∈ fst ` fset xs ∧ root t = y))
    case True
    have disjoint-darcs (child2 y (remove-child y xs) xs)
      using disjoint-darcs-child2[OF wf-arcs] by simp
    moreover have ∀ (x,e) ∈ fset (child2 y (remove-child y xs) xs). wf-darcs x
      using wf-darcs-child2 wf-arcs by fast
    ultimately show ?thesis using True by (simp add: wf-darcs-iff-darcs')
  next
    case False
    have disjoint-darcs ((λ(t,e). (combine x y t, e)) |`| xs)
      using disjoint-darcs-combine ind.prems by simp
    moreover have ∀ (x,e) ∈ fset xs. list-dtree x using list-dtree-rec by blast
   ultimately show ?thesis using False ind.IH ind.prems by (auto simp: wf-darcs-iff-darcs')
  qed
qed

lemma v-in-dlverts-if-in-comb: v ∈ dlverts (combine x y t) ⟹ v ∈ dlverts t
using list-dtree-axioms proof(induction t)
  case ind: (Node r xs)
  then interpret list-dtree Node r xs using ind.prems by blast
  show ?case
  proof(cases x=r ∧ (∃ t. t ∈ fst ` fset xs ∧ root t = y))
    case x-and-y: True
    show ?thesis
    proof(cases v ∈ set x ∪ set y)
      case True
     then show ?thesis using x-and-y dtree.set-sel(1) lverts-if-in-verts by fastforce
    next
      case False
```

```
    then obtain t e where t-def: (t,e) ∈ fset (child2 y (remove-child y xs) xs) v
∈ dlverts t
      using x-and-y ind.prems by auto
    then show ?thesis
    proof(cases (t,e) |∈| (remove-child y xs))
      case True
      then have (t,e) ∈ fset (remove-child y xs) by fast
      then show ?thesis using t-def(2) by force
    next
      case False
       then obtain r1 rs1 re1 where r1-def: (Node r1 rs1, re1) ∈ fset xs (t,e)
|∈| rs1
        using child2-in-child t-def(1) by fast
      have is-subtree t (Node r1 rs1) using subtree-if-child r1-def(2)
        by (metis image-iff prod.sel(1))
      moreover have is-subtree (Node r1 rs1) (Node r xs)
        using subtree-if-child r1-def(1) by fastforce
      ultimately have is-subtree t (Node r xs) using subtree-trans by blast
      then show ?thesis using t-def(2) subtree-in-dlverts by blast
    qed
  qed
 next
  case rec: False
  then show ?thesis
  proof(cases v ∈ set r)
    case False
    then have ∃(t,e) ∈ fset xs. v ∈ dlverts (combine x y t)
      using ind.prems list-dtree-rec rec by force
    then show ?thesis using ind.IH list-dtree-rec by fastforce
  qed (simp)
 qed
qed

lemma ex-subtree-if-in-lverts: v ∈ dlverts t1 ⟹ ∃ t2. is-subtree t2 t1 ∧ v ∈ set
(root t2)
 apply(induction t1)
 apply(cases)
  apply simp
 by fastforce

lemma child'-in-child2:
 assumes (Node y rs1,e1) ∈ fset xs and (t2,e2) ∈ fset rs1
 shows (t2,e2) ∈ fset (child2 y ys xs)
using assms proof(induction xs)
 case (insert x xs)
 obtain r rs re where r-def: (Node r rs, re) = x by (metis dtree.exhaust surj-pair)
 show ?case
 proof(cases r = y)
   case ry: True
```

```
    then have 0: child2 y ys (finsert x xs) = rs |∪| (child2 y ys xs)
      using r-def insert.hyps(1) by force
    then show ?thesis using insert by fastforce
  next
    case False
    then show ?thesis using insert r-def by force
  qed
qed (simp)

lemma v-in-comb-if-in-dlverts: v ∈ dlverts t ⟹ v ∈ dlverts (combine x y t)
using list-dtree-axioms proof(induction t)
  case ind: (Node r xs)
  then interpret list-dtree Node r xs using ind.prems by blast
  show ?case
  proof(cases x=r ∧ (∃ t. t ∈ fst ‘ fset xs ∧ root t = y))
    case x-and-y: True
    then have 0: combine x y (Node r xs) = Node (x@y) (child2 y (remove-child
y xs) xs) by simp
    show ?thesis
    proof(cases v ∈ set x ∪ set y)
      case True
     then show ?thesis using x-and-y dtree.set-sel(1) lverts-if-in-verts by fastforce
    next
      case False
      obtain t where t-def: is-subtree t (Node r xs) v ∈ set (root t)
        using ex-subtree-if-in-lverts ind.prems by fast
      then have Node r xs ≠ t using False x-and-y by fastforce
      then obtain t1 e1 where t1-def: is-subtree t t1 (t1,e1) ∈ fset xs
        using t-def(1) by force
      then show ?thesis
      proof(cases root t1 = y)
        case True
        then have t1 ≠ t using False t-def(2) by blast
      then obtain rs1 where rs1-def: t1 = Node y rs1 using True dtree.exhaust-sel
by blast
        then obtain t2 e2 where t2-def: is-subtree t t2 (t2,e2) ∈ fset rs1
          using ‹t1≠t› t1-def(1) by auto
        have (t2,e2) ∈ fset (child2 y (remove-child y xs) xs)
          using t2-def(2) rs1-def t1-def(2) child'-in-child2 by fast
        then have is-subtree t2 (combine x y (Node r xs)) using subtree-if-child 0
          using self-subtree by fastforce
         then have is-subtree t (combine x y (Node r xs)) using subtree-trans
t2-def(1) by blast
        then show ?thesis
         using t-def(2) t2-def(1) subtree-in-dlverts dtree.set-sel(1) lverts-if-in-verts
by fast
      next
        case False
        then have (t1,e1) ∈ fset (remove-child y xs) using t1-def(2) by simp
```

```
      then have (t1,e1) ∈ fset (child2 y (remove-child y xs) xs)
        using less-eq-fset.rep-eq input-in-child2 by fast
      then have is-subtree t (combine x y (Node r xs))
        using 0 subtree-if-child subtree-trans t1-def(1) by auto
      then show ?thesis
        using t-def(2) subtree-in-dlverts dtree.set-sel(1) lverts-if-in-verts by fast
    qed
  qed
 next
  case rec: False
  then show ?thesis
  proof(cases v ∈ set r)
    case False
    then obtain t e where t-def: (t,e) ∈ fset xs v ∈ dlverts t using ind.prems
by auto
    then have v ∈ dlverts (combine x y t) using ind.IH list-dtree-rec by auto
    then show ?thesis using rec t-def(1) by force
  qed (simp)
 qed
qed

lemma dlverts-comb-id[simp]: dlverts (combine x y t) = dlverts t
 using v-in-comb-if-in-dlverts v-in-dlverts-if-in-comb by blast

lemma wf-dlverts-comb-aux:
 assumes ∀(t,e) ∈ fset xs. dlverts (combine x y t) = dlverts t
     and ∀(t1,e1) ∈ fset xs. ∀(t2,e2) ∈ fset xs. dlverts t1 ∩ dlverts t2 = {} ∨
(t1,e1)=(t2,e2)
     and (t1,e1) ∈ fset ((λ(t,e). (combine x y t, e)) |`| xs)
     and (t2,e2) ∈ fset ((λ(t,e). (combine x y t, e)) |`| xs)
   shows dlverts t1 ∩ dlverts t2 = {} ∨ (t1,e1)=(t2,e2)
proof −
 obtain t1' where t1-def: combine x y t1' = t1 (t1',e1) ∈ fset xs using assms(3)
by auto
 obtain t2' where t2-def: combine x y t2' = t2 (t2',e2) ∈ fset xs using assms(4)
by auto
 show ?thesis
 proof(cases dlverts t1' ∩ dlverts t2' = {})
   case True
   then show ?thesis using assms(1) t1-def t2-def by blast
 next
   case False
   then show ?thesis using assms(2) t1-def t2-def by fast
 qed
qed

lemma wf-dlverts-child2:
 assumes (t1,e) ∈ fset (child2 y (remove-child y xs) xs)
     and ∀(t,e) ∈ fset xs. wf-dlverts t
```

**shows** *wf-dlverts t1*
**proof**(*cases* (*t1*,*e*) |∈| (*remove-child y xs*))
**case** *True*
**then show** *?thesis* **using** *assms*(*2*) **by** *fastforce*
**next**
**case** *False*
**then obtain** *rs re* **where** *r-def*: (*Node y rs*, *re*) ∈ *fset xs* (*t1*,*e*)|∈| *rs*
**using** *child2-in-child assms*(*1*) **by** *fast*
**then show** *?thesis* **using** *assms*(*2*) **by** *fastforce*
**qed**

**lemma** *wf-dlverts-child2-aux1*:
**assumes** (*t1*,*e1*) ∈ *fset* (*child2 y* (*remove-child y xs*) *xs*)
**and** ∃ *t*. *t* ∈ *fst* ‘ *fset xs* ∧ *root t* = *y*
**and** *wf-dlverts* (*Node r xs*)
**shows** *set* (*r*@*y*) ∩ *dlverts t1* = {}
**proof**(*cases* (*t1*,*e1*) |∈| (*remove-child y xs*))
**case** *True*
**then have** *t1-def*: *root t1* ≠ *y* (*t1*,*e1*) ∈ *fset xs* **by** *fastforce*+
**obtain** *t et* **where** *t-def*: (*t*,*et*) ∈ *fset xs root t* = *y* **using** *assms*(*2*) **by** *force*
**have** ∀ *y*′∈ *set y*. *y*′ ∉ *dlverts t1*
**proof**
**fix** *y*′
**assume** *y*′ ∈ *set y*
**then have** *asm*: *y*′ ∈ *dlverts t* **using** *t-def*(*2*) *dtree.set-sel*(*1*) *lverts-if-in-verts*
**by** *fastforce*
**have** *dlverts t1* ∩ *dlverts t* = {} **using** *assms*(*3*) *t1-def t-def* **by** *fastforce*
**then show** *y*′ ∉ *dlverts t1* **using** *asm* **by** *blast*
**qed**
**then show** *?thesis* **using** *assms*(*3*) *t1-def*(*2*) **by** *auto*
**next**
**case** *False*
**then obtain** *rs1 re1* **where** *r-def*: (*Node y rs1*, *re1*) ∈ *fset xs* (*t1*,*e1*)|∈| *rs1*
**using** *child2-in-child assms*(*1*) **by** *fast*
**have** ∀ *y*′∈ *set y*. *y*′ ∉ *dlverts t1* **using** *assms*(*3*) *r-def* **by** *fastforce*
**then show** *?thesis* **using** *assms*(*3*) *r-def* **by** *fastforce*
**qed**

**lemma** *wf-dlverts-child2-aux2*:
**assumes** ∀ (*t1*,*e1*) ∈ *fset xs*. ∀ (*t2*,*e2*) ∈ *fset xs*. *dlverts t1* ∩ *dlverts t2* = {} ∨
(*t1*,*e1*)=(*t2*,*e2*)
**and** ∀ (*t*,*e*) ∈ *fset xs*. *wf-dlverts t*
**and** (*t1*,*e1*) ∈ *fset* (*child2 y* (*remove-child y xs*) *xs*)
**and** (*t2*,*e2*) ∈ *fset* (*child2 y* (*remove-child y xs*) *xs*)
**and** (*t1*,*e1*)≠(*t2*,*e2*)
**shows** *dlverts t1* ∩ *dlverts t2* = {}
**proof**(*cases* (*t1*,*e1*) |∈| (*remove-child y xs*))
**case** *t1-r*: *True*
**then show** *?thesis*

```
  proof(cases (t2,e2) |∈| (remove-child y xs))
    case True
    then show ?thesis
      by (smt (verit, ccfv-threshold) t1-r assms(1,5) Int-iff case-prodD filter-fset)
  next
    case False
    then obtain rs2 re2 where r-def: (Node y rs2, re2) ∈ fset xs (t2,e2)|∈| rs2
      using child2-in-child assms(4) by fast
    then show ?thesis
      using t1-r assms(1) ffmember-filter inf-assoc inf-bot-right inf-commute
        by (smt (verit) dtree.sel(1) semilattice-inf-class.inf.absorb-iff2 case-prodD
child-in-dlverts)
  qed
next
  case False
  then obtain rs1 re1 where r1-def: (Node y rs1, re1) ∈ fset xs (t1,e1)|∈| rs1
    using child2-in-child assms(3) by fast
  show ?thesis
  proof(cases (t2,e2) |∈| (remove-child y xs))
    case True
    then show ?thesis
      using r1-def assms(1) ffmember-filter inf-assoc inf-bot-right inf-commute
        by (smt (verit) dtree.sel(1) semilattice-inf-class.inf.absorb-iff2 case-prodD
child-in-dlverts)
  next
    case False
    then obtain rs2 re2 where r2-def: (Node y rs2, re2) ∈ fset xs (t2,e2) |∈| rs2
      using child2-in-child assms(4) by fast
    then show ?thesis
    proof(cases rs1=rs2)
      case True
      have ∀(t1,e1) ∈ fset rs1. ∀(t2,e2) ∈ fset rs1.
              dlverts t1 ∩ dlverts t2 = {} ∨ (t1,e1)=(t2,e2)
        using r1-def(1) assms(2) by fastforce
      then show ?thesis
        using r1-def(2) r2-def(2) assms(5) True
        by (metis (mono-tags, lifting) case-prodD)
    next
      case False
      then have dlverts (Node y rs1) ∩ dlverts (Node y rs2) = {}
        using assms(1) r1-def(1) r2-def(1) by fast
      then show ?thesis
        using r1-def(2) r2-def(2) child-in-dlverts
        by (metis order-bot-class.bot.extremum-uniqueI inf-mono)
    qed
  qed
qed

lemma wf-dlverts-combine: wf-dlverts (combine x y t)
```

**using** *list-dtree-axioms* **proof**(*induction t*)
**case** *ind*: (*Node r xs*)
**then interpret** *list-dtree Node r xs* **using** *ind.prems* **by** *blast*
**show** *?case*
**proof**(*cases x=r* ∧ (∃ *t*. *t* ∈ *fst* ‘ *fset xs* ∧ *root t* = *y*))
**case** *True*
**let** *?xs* = *child2 y* (*remove-child y xs*) *xs*
**have** ∀ (*t1*,*e1*) ∈ *fset xs*. ∀ (*t2*,*e2*) ∈ *fset xs*.
*dlverts t1* ∩ *dlverts t2* = {} ∨ (*t1*,*e1*)=(*t2*,*e2*) **using** *wf-lverts* **by** *fastforce*
**moreover have** ∀ (*t1*,*e1*) ∈ *fset xs*. *wf-dlverts t1* **using** *wf-lverts* **by** *fastforce*
**ultimately have** ∀ (*t1*,*e1*) ∈ *fset ?xs*. ∀ (*t2*,*e2*) ∈ *fset ?xs*.
*dlverts t1* ∩ *dlverts t2* = {} ∨ (*t1*,*e1*)=(*t2*,*e2*)
**using** *wf-dlverts-child2-aux2*[*of xs*] **by** *blast*
**moreover have** ∀ (*x*,*e*) ∈ *fset ?xs*. *wf-dlverts x* **using** *wf-dlverts-child2 wf-lverts*
**by** *fastforce*
**moreover have** (*x*@*y*) ≠ [] **using** *True wf-lverts* **by** *simp*
**moreover have** ∀ (*t1*,*e1*) ∈ *fset ?xs*. *set* (*x*@*y*) ∩ *dlverts t1* = {}
**using** *wf-dlverts-child2-aux1 wf-lverts True* **by** *fast*
**ultimately have** *wf-dlverts* (*Node* (*x*@*y*) *?xs*) **by** *fastforce*
**moreover have** *combine x y* (*Node r xs*) = *Node* (*x*@*y*) *?xs* **using** *True* **by**
*simp*
**ultimately show** *?thesis* **by** *argo*
**next**
**case** *False*
**let** *?xs* = (λ(*t*,*e*). (*combine x y t*, *e*)) |‘| *xs*
**have** *0*: ∀ (*t*,*e*) ∈ *fset xs*. *dlverts* (*combine x y t*) = *dlverts t*
**using** *list-dtree.dlverts-comb-id list-dtree-rec* **by** *fast*
**have** *1*: ∀ (*t*,*e*) ∈ *fset ?xs*. *wf-dlverts t* **using** *ind.IH list-dtree-rec* **by** *auto*
**have** *2*: ∀ (*t*,*e*) ∈ *fset ?xs*. *set r* ∩ *dlverts t* = {} **using** *0 wf-lverts* **by** *fastforce*
**have** ∀ (*t1*,*e1*) ∈ *fset xs*. ∀ (*t2*,*e2*) ∈ *fset xs*.
*dlverts t1* ∩ *dlverts t2* = {} ∨ (*t1*,*e1*)=(*t2*,*e2*) **using** *wf-lverts* **by** *fastforce*
**then have** *3*: ∀ (*t1*,*e1*) ∈ *fset ?xs*. ∀ (*t2*,*e2*) ∈ *fset ?xs*.
*dlverts t1* ∩ *dlverts t2* = {} ∨ (*t1*,*e1*)=(*t2*,*e2*)
**using** *0 wf-dlverts-comb-aux*[*of xs*] **by** *blast*
**have** *4*: *combine x y* (*Node r xs*) = *Node r ?xs* **using** *False* **by** *auto*
**have** *r* ≠ [] **using** *wf-lverts* **by** *simp*
**then show** *?thesis* **using** *1 2 3 4* **by** *fastforce*
**qed**
**qed**

**theorem** *list-dtree-comb*: *list-dtree* (*combine x y t*)
**by**(*unfold-locales*) (*auto simp*: *wf-darcs-combine wf-dlverts-combine*)

**end**

**end**

**theory** *IKKBZ*

**imports** *Complex-Main CostFunctions QueryGraph List-Dtree HOL−Library.Sorting-Algorithms*
**begin**

# 9 IKKBZ

## 9.1 Additional Proofs for Merging Lists

**lemma** *merge-comm-if-not-equiv*: $\forall x \in set\ xs.\ \forall y \in set\ ys.\ compare\ cmp\ x\ y \neq Equiv \Longrightarrow$
*Sorting-Algorithms.merge cmp xs ys = Sorting-Algorithms.merge cmp ys xs*
**apply**(*induction xs ys rule*: *Sorting-Algorithms.merge.induct*)
**by**(*auto intro*: *compare.quasisym-not-greater simp*: *compare.asym-greater*)

**lemma** *set-merge*: *set xs ∪ set ys = set* (*Sorting-Algorithms.merge cmp xs ys*)
**using** *mset-merge set-mset-mset set-mset-union* **by** *metis*

**lemma** *input-empty-if-merge-empty*: *Sorting-Algorithms.merge cmp xs ys = [] ⟹ xs = [] ∧ ys = []*
**using** *Un-empty set-empty2 set-merge* **by** *metis*

**lemma** *merge-assoc*:
*Sorting-Algorithms.merge cmp xs* (*Sorting-Algorithms.merge cmp ys zs*)
*= Sorting-Algorithms.merge cmp* (*Sorting-Algorithms.merge cmp xs ys*) *zs*
(**is** *?merge - xs* (*?merge cmp - zs*) = -)
**proof**(*induction xs ?merge cmp ys zs arbitrary*: *ys zs taking*: *cmp rule*: *Sorting-Algorithms.merge.induct*)
**case** (*2 cmp v vs*)
**show** *?case* **using** *input-empty-if-merge-empty*[*OF 2*[*symmetric*]] **by** *simp*
**next**
**case** *ind*: (*3 x xs r rs*)
**then show** *?case*
**proof**(*induction ys zs taking*: *cmp rule*: *Sorting-Algorithms.merge.induct*)
**case** (*3 y ys z zs*)
**then show** *?case*
**using** *ind compare.asym-greater*
**by** (*smt* (*verit*, *best*) *compare.trans-not-greater list.inject merge.simps*(*3*))
**qed** (*auto*)
**qed** (*simp*)

**lemma** *merge-comp-commute*:
**assumes** $\forall x \in set\ xs.\ \forall y \in set\ ys.\ compare\ cmp\ x\ y \neq Equiv$
**shows** *Sorting-Algorithms.merge cmp xs* (*Sorting-Algorithms.merge cmp ys zs*)
*= Sorting-Algorithms.merge cmp ys* (*Sorting-Algorithms.merge cmp xs zs*)
**using** *assms merge-assoc merge-comm-if-not-equiv* **by** *metis*

**lemma** *wf-list-arcs-merge*:
⟦*wf-list-arcs xs*; *wf-list-arcs ys*; *snd ‘ set xs ∩ snd ‘ set ys = {}*⟧
⟹ *wf-list-arcs* (*Sorting-Algorithms.merge cmp xs ys*)
**proof**(*induction xs ys taking*: *cmp rule*: *Sorting-Algorithms.merge.induct*)

```
  case (3 x xs y ys)
  obtain v1 e1 where v1-def[simp]: x = (v1,e1) by force
  obtain v2 e2 where v2-def[simp]: y = (v2,e2) by force
  show ?case
  proof(cases compare cmp x y = Greater)
    case True
    have e2 ∉ snd ` set (x#xs) using 3.prems(3) by auto
    moreover have e2 ∉ snd ` set ys using 3.prems(2) by simp
    ultimately have e2 ∉ snd ` set (Sorting-Algorithms.merge cmp (x#xs) ys)
      using set-merge by fast
    then show ?thesis using True 3 by force
  next
    case False
    have e1 ∉ snd `set (y#ys) using 3.prems(3) by auto
    moreover have e1 ∉ snd ` set xs using 3.prems(1) by simp
    ultimately have e1 ∉ snd `set (Sorting-Algorithms.merge cmp xs (y#ys))
      using set-merge by fast
    then show ?thesis using False 3 by force
  qed
qed (auto)

lemma wf-list-lverts-merge:
  ⟦wf-list-lverts xs; wf-list-lverts ys;
    ∀ v1 ∈ fst ` set xs. ∀ v2 ∈ fst ` set ys. set v1 ∩ set v2 = {}⟧
    ⟹ wf-list-lverts (Sorting-Algorithms.merge cmp xs ys)
proof(induction xs ys taking: cmp rule: Sorting-Algorithms.merge.induct)
  case (3 x xs y ys)
  obtain v1 e1 where v1-def[simp]: x = (v1,e1) by force
  obtain v2 e2 where v2-def[simp]: y = (v2,e2) by force
  show ?case
  proof(cases compare cmp x y = Greater)
    case True
    have ∀ v ∈ fst ` set (x#xs). set v2 ∩ set v = {} using 3.prems(3) by auto
    moreover have ∀ v ∈ fst ` set ys. set v2 ∩ set v = {} using 3.prems(2) by
simp
   ultimately have ∀ v ∈ fst ` set (Sorting-Algorithms.merge cmp (x#xs) ys). set
v2 ∩ set v = {}
      using set-merge[of x#xs] by blast
    then show ?thesis using True 3 by force
  next
    case False
    have ∀ v ∈ fst ` set (y#ys). set v1 ∩ set v = {} using 3.prems(3) by auto
    moreover have ∀ v ∈ fst ` set xs. set v1 ∩ set v = {} using 3.prems(1) by
simp
   ultimately have ∀ v ∈ fst ` set (Sorting-Algorithms.merge cmp xs (y#ys)). set
v1 ∩ set v = {}
      using set-merge[of xs] by auto
    then show ?thesis using False 3 by force
  qed
```

**qed** (*auto*)

**lemma** *merge-hd-exists-preserv*:
⟦∃ (*t1*,*e1*) ∈ *fset xs*. *hd as* = (*root t1*,*e1*); ∃ (*t1*,*e1*) ∈ *fset xs*. *hd bs* = (*root t1*,*e1*)⟧
⟹ ∃ (*t1*,*e1*) ∈ *fset xs*. *hd* (*Sorting-Algorithms.merge cmp as bs*) = (*root t1*,*e1*)
**by**(*induction as bs rule*: *Sorting-Algorithms.merge.induct*) *auto*

**lemma** *merge-split-supset*:
**assumes** *as*@*r*#*bs* = (*Sorting-Algorithms.merge cmp xs ys*)
**shows** ∃ *bs*′ *as*′. *set bs*′ ⊆ *set bs* ∧ (*as*′@*r*#*bs*′ = *xs* ∨ *as*′@*r*#*bs*′ = *ys*)
**using** *assms* **proof**(*induction xs ys arbitrary*: *as taking*: *cmp rule*: *Sorting-Algorithms.merge.induct*)
**case** (*3 x xs y ys*)
**let** *?merge* = *Sorting-Algorithms.merge cmp*
**show** *?case*
**proof**(*cases compare cmp x y* = *Greater*)
**case** *True*
**then show** *?thesis*
**proof**(*cases as*)
**case** *Nil*
**have** *set ys* ⊆ *set* (*?merge* (*x*#*xs*) *ys*) **using** *set-merge* **by** *fast*
**then show** *?thesis* **using** *Nil True 3.prems* **by** *auto*
**next**
**case** (*Cons c cs*)
**then have** *cs*@*r*#*bs* = *?merge* (*x*#*xs*) *ys* **using** *True 3.prems* **by** *simp*
**then obtain** *as*′ *bs*′ **where** *as-def*: *set bs*′ ⊆ *set bs as*′@*r*#*bs*′ = *x*#*xs* ∨ *as*′@*r*#*bs*′ = *ys*
**using** *3.IH*(*1*)[*OF True*] **by** *blast*
**have** *as*′@*r*#*bs*′ = *x*#*xs* ∨ (*y*#*as*′)@*r*#*bs*′ = *y*#*ys* **using** *as-def*(*2*) **by** *simp*
**then show** *?thesis* **using** *as-def*(*1*) **by** *blast*
**qed**
**next**
**case** *False*
**then show** *?thesis*
**proof**(*cases as*)
**case** *Nil*
**have** *set xs* ⊆ *set* (*?merge xs* (*y*#*ys*)) **using** *set-merge* **by** *fast*
**then show** *?thesis* **using** *Nil False 3.prems* **by** *auto*
**next**
**case** (*Cons c cs*)
**then have** *cs*@*r*#*bs* = *?merge xs* (*y*#*ys*) **using** *False 3.prems* **by** *simp*
**then obtain** *as*′ *bs*′ **where** *as-def*: *set bs*′ ⊆ *set bs as*′@*r*#*bs*′ = *xs* ∨ *as*′@*r*#*bs*′ = *y*#*ys*
**using** *3.IH*(*2*)[*OF False*] **by** *blast*
**have** (*x*#*as*′)@*r*#*bs*′ = *x*#*xs* ∨ *as*′@*r*#*bs*′ = *y*#*ys* **using** *as-def*(*2*) **by** *simp*
**then show** *?thesis* **using** *as-def*(*1*) **by** *blast*
**qed**
**qed**
**qed**(*auto*)

**lemma** *merge-split-supset-fst*:
**assumes** $as@(r,e)\#bs = (Sorting\text{-}Algorithms.merge\ cmp\ xs\ ys)$
**shows** $\exists as'\ bs'.\ set\ bs' \subseteq set\ bs \wedge (as'@(r,e)\#bs' = xs \vee as'@(r,e)\#bs' = ys)$
**using** *merge-split-supset*[*OF assms*] **by** *blast*

**lemma** *merge-split-supset'*:
**assumes** $r \in set\ (Sorting\text{-}Algorithms.merge\ cmp\ xs\ ys)$
**shows** $\exists as\ bs\ as'\ bs'.\ as@r\#bs = (Sorting\text{-}Algorithms.merge\ cmp\ xs\ ys)$
$\wedge\ set\ bs' \subseteq set\ bs \wedge (as'@r\#bs' = xs \vee as'@r\#bs' = ys)$
**using** *merge-split-supset split-list*[*OF assms*] **by** *metis*

**lemma** *merge-split-supset-fst'*:
**assumes** $r \in fst\ `\ set\ (Sorting\text{-}Algorithms.merge\ cmp\ xs\ ys)$
**shows** $\exists as\ e\ bs\ as'\ bs'.\ as@(r,e)\#bs = (Sorting\text{-}Algorithms.merge\ cmp\ xs\ ys)$
$\wedge\ set\ bs' \subseteq set\ bs \wedge (as'@(r,e)\#bs' = xs \vee as'@(r,e)\#bs' = ys)$
**proof** −
**obtain** $e$ **where** $(r,e) \in set\ (Sorting\text{-}Algorithms.merge\ cmp\ xs\ ys)$ **using** *assms*
**by** *auto*
**then show** *?thesis* **using** *merge-split-supset'*[*of* $(r,e)$] **by** *blast*
**qed**

**lemma** *merge-split-supset-subtree*:
**assumes** $\forall as\ bs.\ as@(r,e)\#bs = xs \longrightarrow$
$(\exists zs.\ is\text{-}subtree\ (Node\ r\ zs)\ t \wedge dverts\ (Node\ r\ zs) \subseteq fst\ `\ set\ ((r,e)\#bs))$
**and** $\forall as\ bs.\ as@(r,e)\#bs = ys \longrightarrow$
$(\exists zs.\ is\text{-}subtree\ (Node\ r\ zs)\ t \wedge dverts\ (Node\ r\ zs) \subseteq fst\ `\ set\ ((r,e)\#bs))$
**and** $as@(r,e)\#bs = (Sorting\text{-}Algorithms.merge\ cmp\ xs\ ys)$
**shows** $\exists zs.\ is\text{-}subtree\ (Node\ r\ zs)\ t \wedge dverts\ (Node\ r\ zs) \subseteq (fst\ `\ set\ ((r,e)\#bs))$
**proof** −
**obtain** $as'\ bs'$ **where** *bs'-def*: $set\ bs' \subseteq set\ bs\ as'@(r,e)\#bs' = xs \vee as'@(r,e)\#bs' = ys$
**using** *merge-split-supset*[*OF assms*(*3*)] **by** *blast*
**obtain** *zs* **where** *zs-def*: *is-subtree* (*Node r zs*) *t dverts* (*Node r zs*) $\subseteq$ *fst* ‘ *set* ((*r*,*e*)#*bs'*)
**using** *assms*(*1*,*2*) *bs'-def*(*2*) **by** *blast*
**then have** *dverts* (*Node r zs*) $\subseteq$ *fst* ‘ *set* ((*r*,*e*)#*bs*) **using** *bs'-def*(*1*) **by** *auto*
**then show** *?thesis* **using** *zs-def*(*1*) **by** *blast*
**qed**

**lemma** *merge-split-supset-strict-subtree*:
**assumes** $\forall as\ bs.\ as@(r,e)\#bs = xs \longrightarrow (\exists zs.\ strict\text{-}subtree\ (Node\ r\ zs)\ t$
$\wedge\ dverts\ (Node\ r\ zs) \subseteq fst\ `\ set\ ((r,e)\#bs))$
**and** $\forall as\ bs.\ as@(r,e)\#bs = ys \longrightarrow (\exists zs.\ strict\text{-}subtree\ (Node\ r\ zs)\ t$
$\wedge\ dverts\ (Node\ r\ zs) \subseteq fst\ `\ set\ ((r,e)\#bs))$
**and** $as@(r,e)\#bs = (Sorting\text{-}Algorithms.merge\ cmp\ xs\ ys)$
**shows** $\exists zs.\ strict\text{-}subtree\ (Node\ r\ zs)\ t$
$\wedge\ dverts\ (Node\ r\ zs) \subseteq (fst\ `\ set\ ((r,e)\#bs))$
**proof** −
**obtain** $as'\ bs'$ **where** *bs'-def*: $set\ bs' \subseteq set\ bs\ as'@(r,e)\#bs' = xs \vee as'@(r,e)\#bs'$

$= ys$
  **using** *merge-split-supset*[*OF assms*(*3*)] **by** *blast*
 **obtain** *zs* **where** *zs-def*:
    *strict-subtree* (*Node r zs*) *t dverts* (*Node r zs*) ⊆ *fst* ‘ *set* ((*r*,*e*)#*bs*′)
  **using** *assms*(*1*,*2*) *bs*′-*def*(*2*) **by** *blast*
 **then have** *dverts* (*Node r zs*) ⊆ *fst* ‘ *set* ((*r*,*e*)#*bs*) **using** *bs*′-*def*(*1*) **by** *auto*
 **then show** *?thesis* **using** *zs-def*(*1*,*2*) **by** *blast*
**qed**

**lemma** *sorted-app-l*: *sorted cmp* (*xs*@*ys*) ⟹ *sorted cmp xs*
 **by**(*induction xs rule*: *sorted.induct*) *auto*

**lemma** *sorted-app-r*: *sorted cmp* (*xs*@*ys*) ⟹ *sorted cmp ys*
 **by**(*induction xs*) (*auto simp*: *sorted-Cons-imp-sorted*)

## 9.2 Merging Subtrees of Ranked Dtrees

**locale** *ranked-dtree* = *list-dtree t* **for** *t* :: (′*a list*,′*b*) *dtree* +
 **fixes** *rank* :: ′*a list* ⇒ *real*
 **fixes** *cmp* :: (′*a list*×′*b*) *comparator*
 **assumes** *cmp-antisym*:
   ⟦*v1* ≠ []; *v2* ≠ []; *compare cmp* (*v1*,*e1*) (*v2*,*e2*) = *Equiv*⟧ ⟹ *set v1* ∩ *set v2* ≠ {} ∨ *e1*=*e2*
**begin**

**lemma** *ranked-dtree-rec*: ⟦*Node r xs* = *t*; (*x*,*e*) ∈ *fset xs*⟧ ⟹ *ranked-dtree x cmp*
 **using** *wf-arcs wf-lverts* **by**(*unfold-locales*) (*auto dest*: *cmp-antisym*)

**lemma** *ranked-dtree-rec-suc*: (*x*,*e*) ∈ *fset* (*sucs t*) ⟹ *ranked-dtree x cmp*
 **using** *ranked-dtree-rec*[*of root t*] **by** *force*

**lemma** *ranked-dtree-subtree*: *is-subtree x t* ⟹ *ranked-dtree x cmp*
**using** *ranked-dtree-axioms* **proof**(*induction t*)
 **case** (*Node r xs*)
 **then interpret** *ranked-dtree Node r xs* **by** *blast*
 **show** *?case* **using** *Node ranked-dtree-rec* **by** (*cases x* = *Node r xs*) *auto*
**qed**

### 9.2.1 Definitions

**lift-definition** *cmp*′ :: (′*a list*×′*b*) *comparator* **is**
 (λ*x y*. *if rank* (*rev* (*fst x*)) < *rank* (*rev* (*fst y*)) *then Less*
   *else if rank* (*rev* (*fst x*)) > *rank* (*rev* (*fst y*)) *then Greater*
   *else compare cmp x y*)
  **by** (*smt* (*z3*) *comp.distinct*(*3*) *compare.less-iff-sym-greater compare.refl compare.trans-equiv*
     *compare.trans-less comparator-def*)

**abbreviation** *disjoint-sets* :: ((′*a list*, ′*b*) *dtree* × ′*b*) *fset* ⇒ *bool* **where**

*disjoint-sets xs* ≡ *disjoint-darcs xs* ∧ *disjoint-dlverts xs* ∧ (∀ (*t*,*e*) ∈ *fset xs*. [] ∉ *dverts t*)

**abbreviation** *merge-f* :: *'a list* ⇒ ((*'a list*, *'b*) *dtree* × *'b*) *fset*
⇒ (*'a list*, *'b*) *dtree* × *'b* ⇒ (*'a list* × *'b*) *list* ⇒ (*'a list* × *'b*) *list* **where**
*merge-f r xs* ≡ λ(*t*,*e*) *b*. *if* (*t*,*e*) ∈ *fset xs* ∧ *list-dtree* (*Node r xs*)
∧ (∀ (*v*,*e'*) ∈ *set b*. *set v* ∩ *dlverts t* = {} ∧ *v* ≠ [] ∧ *e'* ∉ *darcs t* ∪ {*e*})
*then Sorting-Algorithms.merge cmp'* (*dtree-to-list* (*Node r* {|(*t*,*e*)|})) *b else b*

**definition** *merge* :: (*'a list*,*'b*) *dtree* ⇒ (*'a list*,*'b*) *dtree* **where**
*merge t1* ≡ *dtree-from-list* (*root t1*) (*ffold* (*merge-f* (*root t1*) (*sucs t1*)) [] (*sucs t1*))

### 9.2.2 Commutativity Proofs

**lemma** *cmp-sets-not-dsjnt-if-equiv*:
⟦*v1* ≠ []; *v2* ≠ []⟧ ⟹ *compare cmp'* (*v1*,*e1*) (*v2*,*e2*) = *Equiv* ⟹ *set v1* ∩ *set v2* ≠ {} ∨ *e1*=*e2*
**by**(*auto simp*: *cmp'.rep-eq dest*: *cmp-antisym split*: *if-splits*)

**lemma** *dtree-to-list-x-in-dverts*:
*x* ∈ *fst* ‘ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})) ⟹ *x* ∈ *dverts t1*
**using** *dtree-to-list-sub-dverts-ins* **by** *auto*

**lemma** *dtree-to-list-x-in-dlverts*:
*x* ∈ *fst* ‘ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})) ⟹ *set x* ⊆ *dlverts t1*
**using** *dtree-to-list-x-in-dverts lverts-if-in-verts* **by** *fast*

**lemma** *dtree-to-list-x1-disjoint*:
*dlverts t1* ∩ *dlverts t2* = {}
⟹ ∀ *x1* ∈ *fst* ‘ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})). *set x1* ∩ *dlverts t2* = {}
**using** *dtree-to-list-x-in-dlverts* **by** *fast*

**lemma** *dtree-to-list-xs-disjoint*:
*dlverts t1* ∩ *dlverts t2* = {}
⟹ ∀ *x1* ∈ *fst* ‘ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})).
∀ *x2* ∈ *fst* ‘ *set* (*dtree-to-list* (*Node r'* {|(*t2*,*e2*)|})). *set x1* ∩ *set x2* = {}
**using** *dtree-to-list-x-in-dlverts* **by** (*metis inf-mono subset-empty*)

**lemma** *dtree-to-list-e-in-darcs*:
*e* ∈ *snd* ‘ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})) ⟹ *e* ∈ *darcs t1* ∪ {*e1*}
**using** *dtree-to-list-sub-darcs* **by** *fastforce*

**lemma** *dtree-to-list-e-disjoint*:
(*darcs t1* ∪ {*e1*}) ∩ (*darcs t2* ∪ {*e2*}) = {}
⟹ ∀ *e* ∈ *snd* ‘ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})). *e* ∉ *darcs t2* ∪ {*e2*}
**using** *dtree-to-list-e-in-darcs* **by** *fast*

**lemma** *dtree-to-list-es-disjoint*:
(*darcs t1* ∪ {*e1*}) ∩ (*darcs t2* ∪ {*e2*}) = {}
⟹ ∀ *e3* ∈ *snd* ‘ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})).
∀ *e4* ∈ *snd* ‘ *set* (*dtree-to-list* (*Node r′* {|(*t2*,*e2*)|})). *e3* ≠ *e4*
**using** *dtree-to-list-e-disjoint dtree-to-list-e-in-darcs* **by** *fast*

**lemma** *dtree-to-list-xs-not-equiv*:
**assumes** *dlverts t1* ∩ *dlverts t2* = {}
**and** (*darcs t1* ∪ {*e3*}) ∩ (*darcs t2* ∪ {*e4*}) = {}
**and** (*x1*,*e1*) ∈ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e3*)|})) **and** *x1* ≠ []
**and** (*x2*,*e2*) ∈ *set* (*dtree-to-list* (*Node r′* {|(*t2*,*e4*)|})) **and** *x2* ≠ []
**shows** *compare cmp′* (*x1*,*e1*) (*x2*,*e2*) ≠ *Equiv*
**using** *dtree-to-list-xs-disjoint*[*OF assms*(*1*)] *cmp-sets-not-dsjnt-if-equiv*[*of x1 x2 e1 e2*]
*dtree-to-list-es-disjoint*[*OF assms*(*2*)] *assms*(*3*−*6*) **by** *fastforce*

**lemma** *merge-dtree1-not-equiv*:
**assumes** *dlverts t1* ∩ *dlverts t2* = {}
**and** (*darcs t1* ∪ {*e1*}) ∩ (*darcs t2* ∪ {*e2*}) = {}
**and** [] ∉ *dverts t1*
**and** [] ∉ *dverts t2*
**and** *xs* = *dtree-to-list* (*Node r* {|(*t1*,*e1*)|})
**and** *ys* = *dtree-to-list* (*Node r′* {|(*t2*,*e2*)|})
**shows** ∀ (*x1*,*e1*)∈*set xs*. ∀ (*x2*,*e2*)∈*set ys*. *compare cmp′* (*x1*,*e1*) (*x2*,*e2*) ≠ *Equiv*
**proof** −
**have** ∀ (*x1*,*e1*)∈*set xs*. *x1* ≠ []
**using** *assms*(*3*,*5*) *dtree-to-list-x-in-dverts*
**by** (*smt* (*verit*) *case-prod-conv case-prod-eta fst-conv pair-imageI surj-pair*)
**moreover have** ∀ (*x1*,*e1*)∈*set ys*. *x1* ≠ []
**using** *assms*(*4*,*6*) *dtree-to-list-x-in-dverts*
**by** (*smt* (*verit*) *case-prod-conv case-prod-eta fst-conv pair-imageI surj-pair*)
**ultimately show** *?thesis* **using** *dtree-to-list-xs-not-equiv*[*of t1 t2*] *assms*(*1*,*2*,*5*,*6*)
**by** *fast*
**qed**

**lemma** *merge-commute-aux1*:
**assumes** *dlverts t1* ∩ *dlverts t2* = {}
**and** (*darcs t1* ∪ {*e1*}) ∩ (*darcs t2* ∪ {*e2*}) = {}
**and** [] ∉ *dverts t1*
**and** [] ∉ *dverts t2*
**and** *xs* = *dtree-to-list* (*Node r* {|(*t1*,*e1*)|})
**and** *ys* = *dtree-to-list* (*Node r′* {|(*t2*,*e2*)|})
**shows** *Sorting-Algorithms.merge cmp′ xs ys* = *Sorting-Algorithms.merge cmp′ ys xs*
**using** *merge-dtree1-not-equiv merge-comm-if-not-equiv assms* **by** *fast*

**lemma** *dtree-to-list-x1-list-disjoint*:
*set x2* ∩ *dlverts t1* = {}

$\Longrightarrow \forall\, x1 \in fst\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\}))\ .\ set\ x1 \cap set\ x2 = \{\}$
**using** *dtree-to-list-x-in-dlverts* **by** *fast*

**lemma** *dtree-to-list-e1-list-disjoint'*:
$set\ x2 \cap darcs\ t1 \cup \{e1\} = \{\}$
$\Longrightarrow \forall\, x1 \in snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\}))\ .\ x1 \notin set\ x2$
**using** *dtree-to-list-e-in-darcs* **by** *blast*

**lemma** *dtree-to-list-e1-list-disjoint*:
$e2 \notin darcs\ t1 \cup \{e1\}$
$\Longrightarrow \forall\, x1 \in snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\}))\ .\ x1 \neq e2$
**using** *dtree-to-list-e-in-darcs* **by** *fast*

**lemma** *dtree-to-list-xs-list-not-equiv*:
**assumes** $(x1,e1) \in set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e3)|\}))$
**and** $x1 \neq []$
**and** $\forall\,(v,e) \in set\ ys\ .\ set\ v \cap dlverts\ t1 = \{\} \wedge v \neq [] \wedge e \notin darcs\ t1 \cup \{e3\}$
**and** $(x2,e2) \in set\ ys$
**shows** $compare\ cmp'\ (x1,e1)\ (x2,e2) \neq Equiv$
**proof** −
**have** $set\ x1 \cap set\ x2 = \{\}$ **using** *dtree-to-list-x1-list-disjoint assms*$(1,3,4)$ **by** *fastforce*
**moreover have** $e1 \neq e2$ **using** *dtree-to-list-e1-list-disjoint assms*$(1,3,4)$ **by** *fastforce*
**ultimately show** *?thesis* **using** *cmp-sets-not-dsjnt-if-equiv assms*$(2-4)$ **by** *auto*
**qed**

**lemma** *merge-commute-aux2*:
**assumes** $[] \notin dverts\ t1$
**and** $xs = dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\})$
**and** $\forall\,(v,e) \in set\ ys\ .\ set\ v \cap dlverts\ t1 = \{\} \wedge v \neq [] \wedge e \notin darcs\ t1 \cup \{e1\}$
**shows** *Sorting-Algorithms.merge cmp' xs ys = Sorting-Algorithms.merge cmp' ys xs*
**proof** −
**have** $\forall\,(x1,e1) \in set\ xs\ .\ x1 \neq []$
**using** *assms*$(1,2)$ *dtree-to-list-x-in-dverts*
**by** (*smt* (*verit*) *case-prod-conv case-prod-eta fst-conv pair-imageI surj-pair*)
**then have** $\forall\,(x1,e1) \in set\ xs\ .\ \forall\,(x2,e2) \in set\ ys\ .\ compare\ cmp'\ (x1,e1)\ (x2,e2) \neq Equiv$
**using** *assms*$(2,3)$ *dtree-to-list-xs-list-not-equiv* **by** *force*
**then show** *?thesis* **using** *merge-comm-if-not-equiv* **by** *fast*
**qed**

**lemma** *merge-inter-preserv'*:
**assumes** $f = (merge\text{-}f\ r\ xs)$
**and** $\neg(\forall\,(v,\text{-}) \in set\ z\ .\ set\ v \cap dlverts\ t1 = \{\})$
**shows** $\neg(\forall\,(v,\text{-}) \in set\ (f\ (t2,e2)\ z)\ .\ set\ v \cap dlverts\ t1 = \{\})$
**proof**(*cases f* $(t2,e2)\ z = z$)
**case** *False*

**then have** *f (t2,e2) z = Sorting-Algorithms.merge cmp' (dtree-to-list (Node r {|(t2,e2)|})) z*
**by**(*simp add*: *assms(1)*) *meson*
**then show** *?thesis* **using** *assms(2) set-merge* **by** *force*
**qed** (*simp add*: *assms(2)*)

**lemma** *merge-inter-preserv*:
**assumes** *f = (merge-f r xs)*
**and** ¬(∀ *(v,e) ∈ set z. set v ∩ dlverts t1 = {} ∧ e ∉ darcs t1 ∪ {e1}*)
**shows** ¬(∀ *(v,e) ∈ set (f (t2,e2) z). set v ∩ dlverts t1 = {} ∧ e ∉ darcs t1 ∪ {e1}*)
**proof**(*cases f (t2,e2) z = z*)
**case** *True*
**then show** *?thesis* **using** *assms(2)* **by** *simp*
**next**
**case** *False*
**then have** *f (t2,e2) z = Sorting-Algorithms.merge cmp' (dtree-to-list (Node r {|(t2,e2)|})) z*
**by**(*simp add*: *assms(1)*) *meson*
**then show** *?thesis*
**using** *assms(2) set-merge*[*of dtree-to-list (Node r {|(t2,e2)|})*] **by** *simp blast*
**qed**

**lemma** *merge-f-eq-z-if-inter'*:
¬(∀ *(v,-) ∈ set z. set v ∩ dlverts t1 = {}*) ⟹ *(merge-f r xs) (t1,e1) z = z*
**by** *auto*

**lemma** *merge-f-eq-z-if-inter*:
¬(∀ *(v,e) ∈ set z. set v ∩ dlverts t1 = {} ∧ e ∉ darcs t1 ∪ {e1}*)
⟹ *(merge-f r xs) (t1,e1) z = z*
**by** *auto*

**lemma** *merge-empty-inter-preserv-aux*:
**assumes** *f = (merge-f r xs)*
**and** *(t2,e2) ∈ fset xs*
**and** ∀ *(v,e) ∈ set z. set v ∩ dlverts t2 = {} ∧ v≠[] ∧ e ∉ darcs t2 ∪ {e2}*
**and** *list-dtree (Node r xs)*
**and** *(t1,e1) ∈ fset xs*
**and** *(t1,e1) ≠ (t2,e2)*
**and** ∀ *(v,e) ∈ set z. set v ∩ dlverts t1 = {} ∧ v≠[] ∧ e ∉ darcs t1 ∪ {e1}*
**shows** ∀ *(v,e) ∈ set (f (t2,e2) z). set v ∩ dlverts t1 = {} ∧ v≠[] ∧ e ∉ darcs t1 ∪ {e1}*
**proof** −
**have** *0*: *f (t2,e2) z = Sorting-Algorithms.merge cmp' (dtree-to-list (Node r {|(t2,e2)|})) z*
**using** *assms(1−6)* **by** *simp*
**let** *?ys = dtree-to-list (Node r {|(t2,e2)|})*
**interpret** *list-dtree Node r xs* **using** *assms(4)* **.**
**have** *disjoint-dlverts xs* **using** *wf-lverts* **by** *simp*

**then have** $\forall v{\in}fst\ `\ set\ ?ys.\ set\ v \cap dlverts\ t1 = \{\}$
**using** *dtree-to-list-x1-disjoint assms*(2,5,6) **by** *fast*
**then have** *1*: $\forall v{\in}fst\ `\ set\ (Sorting\text{-}Algorithms.merge\ cmp'\ ?ys\ z).\ set\ v \cap dlverts\ t1 = \{\}$
**using** *assms*(7) *set-merge*[*of ?ys*] **by** *fastforce*
**have** *disjoint-darcs xs* **using** *disjoint-darcs-if-wf-xs*[*OF wf-arcs*] **.**
**then have** *2*: $(darcs\ t2 \cup \{e2\}) \cap (darcs\ t1 \cup \{e1\}) = \{\}$ **using** *assms*(2,5,6) **by** *fast*
**have** $\forall e{\in}snd\ `\ set\ ?ys.\ e \notin darcs\ t1 \cup \{e1\}$ **using** *dtree-to-list-e-disjoint*[*OF 2*] **by** *blast*
**then have** *2*: $\forall e{\in}snd\ `\ set\ (Sorting\text{-}Algorithms.merge\ cmp'\ ?ys\ z).\ e \notin darcs\ t1 \cup \{e1\}$
**using** *assms*(7) *set-merge*[*of ?ys*] **by** *fastforce*
**have** $[] \notin dverts\ t2$ **using** *assms*(2) *empty-notin-wf-dlverts wf-lverts* **by** *fastforce*
**then have** $\forall v{\in}fst\ `\ set\ ?ys.\ v \neq []$ **by** (*metis dtree-to-list-x-in-dverts*)
**then have** $\forall v{\in}fst\ `\ set\ (Sorting\text{-}Algorithms.merge\ cmp'\ ?ys\ z).\ v \neq []$
**using** *assms*(7) *set-merge*[*of ?ys*] **by** *fastforce*
**then show** *?thesis* **using** *0 1 2* **by** *fastforce*
**qed**

**lemma** *merge-empty-inter-preserv*:
**assumes** $f = (merge\text{-}f\ r\ xs)$
**and** $\forall (v,e) \in set\ z.\ set\ v \cap dlverts\ t1 = \{\} \wedge v{\neq}[] \wedge e \notin darcs\ t1 \cup \{e1\}$
**and** $(t1,e1) \in fset\ xs$
**and** $(t1,e1) \neq (t2,e2)$
**shows** $\forall (v,e) \in set\ (f\ (t2,e2)\ z).\ set\ v \cap dlverts\ t1 = \{\} \wedge v{\neq}[] \wedge e \notin darcs\ t1 \cup \{e1\}$
**proof**(*cases f* $(t2,e2)\ z = z$)
**case** *True*
**then show** *?thesis* **using** *assms*(2) **by** *simp*
**next**
**case** *False*
**have** $(t2,e2) \in fset\ xs$ **using** *False assms*(1) **by** *simp argo*
**moreover have** *list-dtree* (*Node r xs*) **using** *False assms*(1) **by** *simp argo*
**moreover have** $\forall (v,e) \in set\ z.\ set\ v \cap dlverts\ t2 = \{\} \wedge v{\neq}[] \wedge e \notin darcs\ t2 \cup \{e2\}$
**using** *False assms*(1) **by** *simp argo*
**ultimately show** *?thesis* **using** *merge-empty-inter-preserv-aux assms* **by** *presburger*
**qed**

**lemma** *merge-commute-aux3*:
**assumes** $f = (merge\text{-}f\ r\ xs)$
**and** *list-dtree* (*Node r xs*)
**and** $(t1,e1) \neq (t2,e2)$
**and** $(\forall (v,e) \in set\ z.\ set\ v \cap dlverts\ t1 = \{\} \wedge v \neq [] \wedge e \notin darcs\ t1 \cup \{e1\})$
**and** $(\forall (v,e) \in set\ z.\ set\ v \cap dlverts\ t2 = \{\} \wedge v \neq [] \wedge e \notin darcs\ t2 \cup \{e2\})$
**and** $(t1,e1) \in fset\ xs$
**and** $(t2,e2) \in fset\ xs$

**shows** $(f\ (t2,\ e2) \circ f\ (t1,\ e1))\ z = (f\ (t1,\ e1) \circ f\ (t2,\ e2))\ z$
**proof** −
**let** *?merge* = *Sorting-Algorithms.merge*
**let** *?xs* = *dtree-to-list* (*Node r* {|(*t1*, *e1*)|})
**let** *?ys* = *dtree-to-list* (*Node r* {|(*t2*, *e2*)|})
**interpret** *list-dtree Node r xs* **using** *assms*(*2*) **.**
**have** *disj*: *dlverts t1* ∩ *dlverts t2* = {} [] ∉ *dverts t1* [] ∉ *dverts t2*
**using** *assms*(*3*,*6*,*7*) *disjoint-dlverts-if-wf*[*OF wf-lverts*] *empty-notin-wf-dlverts*[*OF wf-lverts*]
**by** *fastforce*+
**have** *disj2*: (*darcs t1* ∪ {*e1*}) ∩ (*darcs t2* ∪ {*e2*}) = {}
**using** *assms*(*2*,*3*,*6*,*7*) *disjoint-darcs-if-wf-aux5*[*OF wf-arcs*] **by** *blast*
**have** *f* (*t2*, *e2*) *z* = *Sorting-Algorithms.merge cmp′ ?ys z* **using** *assms*(*1*,*2*,*5*,*7*) **by** *simp*
**moreover have** ∀(*v*,*e*)∈*set* (*f* (*t2*,*e2*) *z*). *set v* ∩ *dlverts t1* = {} ∧ *v* ≠ [] ∧ *e* ∉ *darcs t1* ∪ {*e1*}
**using** *merge-empty-inter-preserv*[*OF assms*(*1*)] *assms*(*3*,*4*,*6*) **by** *simp*
**ultimately have** *2*: (*f* (*t1*, *e1*) ∘ *f* (*t2*, *e2*)) *z* = *?merge cmp′ ?xs* (*?merge cmp′ ?ys z*)
**using** *assms*(*1*−*2*,*6*) **by** *auto*
**have** *f* (*t1*, *e1*) *z* = *Sorting-Algorithms.merge cmp′ ?xs z* **using** *assms*(*1*−*2*,*4*,*6*) **by** *simp*
**moreover have** ∀(*v*,*e*)∈*set* (*f* (*t1*, *e1*) *z*). *set v* ∩ *dlverts t2* = {} ∧ *v*≠[] ∧ *e* ∉ *darcs t2* ∪ {*e2*}
**using** *merge-empty-inter-preserv*[*OF assms*(*1*)] *assms*(*3*,*5*,*7*) **by** *presburger*
**ultimately have** *3*: (*f* (*t2*, *e2*) ∘ *f* (*t1*,*e1*)) *z* = *?merge cmp′ ?ys* (*?merge cmp′ ?xs z*)
**using** *assms*(*1*−*2*,*7*) **by** *simp*
**have** ∀ *x*∈*set ?xs*. ∀ *y*∈*set ?ys*. *compare cmp′ x y* ≠ *Equiv*
**using** *merge-dtree1-not-equiv*[*OF disj*(*1*) *disj2*] *disj*(*2*,*3*) **by** *fast*
**then have** *?merge cmp′ ?xs* (*?merge cmp′ ?ys z*) = *?merge cmp′ ?ys* (*?merge cmp′ ?xs z*)
**using** *merge-comp-commute* **by** *blast*
**then show** *?thesis* **using** *2 3* **by** *simp*
**qed**

**lemma** *merge-commute-aux*:
**assumes** *f* = (*merge-f r xs*)
**shows** (*f y* ∘ *f x*) *z* = (*f x* ∘ *f y*) *z*
**proof** −
**obtain** *t1 e1* **where** *y-def*[*simp*]: *x* = (*t1*, *e1*) **by** *fastforce*
**obtain** *t2 e2* **where** *x-def*[*simp*]: *y* = (*t2*, *e2*) **by** *fastforce*
**show** *?thesis*
**proof**(*cases* (*t1*,*e1*) ∈ *fset xs* ∧ (*t2*,*e2*) ∈ *fset xs*)
**case** *True*
**then consider** *list-dtree* (*Node r xs*) (*t1*,*e1*) ≠ (*t2*,*e2*)
(∀(*v*,*e*) ∈ *set z*. *set v* ∩ *dlverts t1* = {} ∧ *v* ≠ [] ∧ *e* ∉ *darcs t1* ∪ {*e1*})
(∀(*v*,*e*) ∈ *set z*. *set v* ∩ *dlverts t2* = {} ∧ *v* ≠ [] ∧ *e* ∉ *darcs t2* ∪ {*e2*})
| (*t1*,*e1*) = (*t2*,*e2*)

```
    | ¬list-dtree (Node r xs)
    | ¬(∀(v,e) ∈ set z. set v ∩ dlverts t1 = {} ∧ e ∉ darcs t1 ∪ {e1})
    | ¬(∀(v,e) ∈ set z. set v ∩ dlverts t2 = {} ∧ e ∉ darcs t2 ∪ {e2})
    | ¬(∀(v,-) ∈ set z. v ≠ [])
    by fast
  then show ?thesis
  proof(cases)
    case 1
    then show ?thesis using merge-commute-aux3[OF assms] True by simp
  next
    case 4
    then have f x z = z by(auto simp: assms)
    then have 0: (f y ∘ f x) z = f y z by simp
    have ¬(∀(v,e) ∈ set (f y z). set v ∩ dlverts t1 = {} ∧ e ∉ darcs t1 ∪ {e1})
      using merge-inter-preserv[OF assms 4] by simp
    then have (f x ∘ f y) z = f y z using assms merge-f-eq-z-if-inter by auto
    then show ?thesis using 0 by simp
  next
    case 5
    then have f y z = z by(auto simp: assms)
    then have 0: (f x ∘ f y) z = f x z by simp
    have ¬(∀(v,e) ∈ set (f x z). set v ∩ dlverts t2 = {} ∧ e ∉ darcs t2 ∪ {e2})
      using merge-inter-preserv[OF assms 5] by simp
    then have (f y ∘ f x) z = f x z using assms merge-f-eq-z-if-inter by simp
    then show ?thesis using 0 by simp
  next
    case 6
    then have (f x ∘ f y) z = z by(auto simp: assms)
    also have z = (f y ∘ f x) z using 6 by(auto simp: assms)
    finally show ?thesis by simp
  qed(auto simp: assms)
 next
  case False
  then have (∀ z. f x z = z) ∨ (∀ z. f y z = z) by(auto simp: assms)
  then show ?thesis by force
 qed
qed

lemma merge-commute: comp-fun-commute (merge-f r xs)
 using comp-fun-commute-def merge-commute-aux by blast

interpretation Comm: comp-fun-commute merge-f r xs by (rule merge-commute)
```

### 9.2.3 Merging Preserves Arcs and Verts

```
lemma empty-list-valid-merge:
 (∀(v,e) ∈ set []. set v ∩ dlverts t1 = {} ∧ v ≠ [] ∧ e ∉ darcs t1 ∪ {e1})
 by simp
```

**lemma** *disjoint-sets-sucs*: *disjoint-sets* (*sucs t*)
**using** *empty-notin-wf-dlverts list-dtree.wf-lverts list-dtree-rec dtree.collapse disjoint-dlverts-if-wf*[*OF wf-lverts*] *disjoint-darcs-if-wf*[*OF wf-arcs*] **by** *blast*

**lemma** *empty-not-elem-subset*:
⟦*xs* |⊆| *ys*; ∀(*t*,*e*) ∈ *fset ys*. [] ∉ *dverts t*⟧ ⟹ ∀(*t*,*e*) ∈ *fset xs*. [] ∉ *dverts t*
**by** (*meson less-eq-fset.rep-eq subset-iff*)

**lemma** *disjoint-sets-subset*:
**assumes** *xs* |⊆| *ys* **and** *disjoint-sets ys*
**shows** *disjoint-sets xs*
**using** *disjoint-darcs-subset*[*OF assms*(*1*)] *disjoint-dlverts-subset*[*OF assms*(*1*)] *empty-not-elem-subset*[*OF assms*(*1*)] *assms* **by** *fast*

**lemma** *merge-mdeg-le-1*: *max-deg* (*merge t1*) ≤ *1*
**unfolding** *merge-def* **by** (*rule dtree-from-list-deg-le-1*)

**lemma** *merge-mdeg-le1-sub*: *is-subtree t1* (*merge t2*) ⟹ *max-deg t1* ≤ *1*
**using** *merge-mdeg-le-1 le-trans mdeg-ge-sub* **by** *fast*

**lemma** *merge-fcard-le1*: *fcard* (*sucs* (*merge t1*)) ≤ *1*
**unfolding** *merge-def* **by** (*rule dtree-from-list-fcard-le1*)

**lemma** *merge-fcard-le1-sub*: *is-subtree t1* (*merge t2*) ⟹ *fcard* (*sucs t1*) ≤ *1*
**using** *merge-mdeg-le1-sub mdeg-ge-fcard*[*of sucs t1 root t1*] **by** *force*

**lemma** *merge-f-alt*:
**assumes** *P* = (λ*xs*. *list-dtree* (*Node r xs*))
**and** *Q* = (λ(*t*,*e*) *b*. (∀(*v*,*e'*) ∈ *set b*. *set v* ∩ *dlverts t* = {} ∧ *v*≠[] ∧ *e'* ∉ *darcs t* ∪ {*e*}))
**and** *R* = (λ(*t*,*e*) *b*. *Sorting-Algorithms.merge cmp'* (*dtree-to-list* (*Node r* {|(*t*,*e*)|})) *b*)
**shows** *merge-f r xs* = (λ*a b*. *if a* ∉ *fset xs* ∨ ¬ *Q a b* ∨ ¬ *P xs then b else R a b*)
**using** *assms* **by** *force*

**lemma** *merge-f-alt-commute*:
**assumes** *P* = (λ*xs*. *list-dtree* (*Node r xs*))
**and** *Q* = (λ(*t*,*e*) *b*. (∀(*v*,*e'*) ∈ *set b*. *set v* ∩ *dlverts t* = {} ∧ *v* ≠ [] ∧ *e'* ∉ *darcs t* ∪ {*e*}))
**and** *R* = (λ(*t*,*e*) *b*. *Sorting-Algorithms.merge cmp'* (*dtree-to-list* (*Node r* {|(*t*,*e*)|})) *b*)
**shows** *comp-fun-commute* (λ*a b*. *if a* ∉ *fset xs* ∨ ¬ *Q a b* ∨ ¬ *P xs then b else R a b*)
**proof** −
**have** *comp-fun-commute* (*merge-f r xs*) **using** *merge-commute* **by** *fast*
**then show** *?thesis* **using** *merge-f-alt*[*OF assms*] **by** *simp*
**qed**

**lemma** *merge-ffold-supset*:
**assumes** *xs* |⊆| *ys* **and** *list-dtree* (*Node r ys*)
**shows** *ffold* (*merge-f r ys*) *acc xs* = *ffold* (*merge-f r xs*) *acc xs*
**proof** −
**let** *?P* = λ*xs*. *list-dtree* (*Node r xs*)
**let** *?Q* = λ(*t*,*e*) *b*. (∀(*v*,*e'*) ∈ *set b*. *set v* ∩ *dlverts t* = {} ∧ *v* ≠ [] ∧ *e'* ∉ *darcs t* ∪ {*e*})
**let** *?R* = λ(*t*,*e*) *b*. *Sorting-Algorithms.merge cmp'* (*dtree-to-list* (*Node r* {|(*t*,*e*)|})) *b*
**have** *0*: ⋀*xs*. *comp-fun-commute* (λ*a b*. *if a* ∉ *fset xs* ∨ ¬ *?Q a b* ∨ ¬ *?P xs then b else ?R a b*)
**using** *merge-f-alt-commute* **by** *blast*
**have** *ffold* (λ*a b*. *if a* ∉ *fset ys* ∨ ¬ *?Q a b* ∨ ¬ *?P ys then b else ?R a b*) *acc xs*
= *ffold* (λ*a b*. *if a* ∉ *fset xs* ∨ ¬ *?Q a b* ∨ ¬ *?P xs then b else ?R a b*) *acc xs*
**using** *ffold-commute-supset*[*OF assms*(*1*), *of ?P ?Q ?R*, *OF assms*(*2*) *list-dtree-subset 0*] **by** *auto*
**then show** *?thesis* **using** *merge-f-alt* **by** *presburger*
**qed**

**lemma** *merge-f-merge-if-not-snd*:
*merge-f r xs* (*t1*,*e1*) *z* ≠ *z* ⟹
*merge-f r xs* (*t1*,*e1*) *z* = *Sorting-Algorithms.merge cmp'* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})) *z*
**by**(*simp*) *meson*

**lemma** *merge-f-merge-if-conds*:
⟦*list-dtree* (*Node r xs*); ∀(*v*,*e*) ∈ *set z*. *set v* ∩ *dlverts t1* = {} ∧ *v*≠[] ∧ *e* ∉ *darcs t1* ∪ {*e1*};
(*t1*,*e1*) ∈ *fset xs*⟧
⟹ *merge-f r xs* (*t1*,*e1*) *z* = *Sorting-Algorithms.merge cmp'* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})) *z*
**by** *force*

**lemma** *merge-f-merge-if-conds-empty*:
⟦*list-dtree* (*Node r xs*); (*t1*,*e1*) ∈ *fset xs*⟧
⟹ *merge-f r xs* (*t1*,*e1*) []
= *Sorting-Algorithms.merge cmp'* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})) []
**using** *merge-f-merge-if-conds* **by** *simp*

**lemma** *merge-ffold-empty-inter-preserv*:
⟦*list-dtree* (*Node r ys*); *xs* |⊆| *ys*;
∀(*v*,*e*) ∈ *set z*. *set v* ∩ *dlverts t1* = {} ∧ *v*≠[] ∧ *e* ∉ *darcs t1* ∪ {*e1*};
(*t1*,*e1*) ∈ *fset ys*; (*t1*,*e1*) ∉ *fset xs*; (*v*,*e*) ∈ *set* (*ffold* (*merge-f r xs*) *z xs*)⟧
⟹ *set v* ∩ *dlverts t1* = {} ∧ *v* ≠ [] ∧ *e* ∉ *darcs t1* ∪ {*e1*}
**proof**(*induction xs*)
**case** (*insert x xs*)
**let** *?f* = *merge-f r* (*finsert x xs*)
**let** *?f'* = *merge-f r xs*
**let** *?merge* = *Sorting-Algorithms.merge*

**interpret** *list-dtree Node r ys* **using** *insert.prems(1)* **.**
**have** *0*: *list-dtree* (*Node r* (*finsert x xs*)) **using** *list-dtree-subset insert.prems(1,2)* **by** *blast*
**show** *?case*
**proof**(*cases ffold ?f z* (*finsert x xs*) = *ffold ?f' z xs*)
**case** *True*
**then have** (*v*,*e*) ∈ *set* (*ffold ?f' z xs*) **using** *insert.prems(6)* **by** *argo*
**then show** *?thesis* **using** *insert.IH insert.prems* **by** *force*
**next**
**case** *not-right*: *False*
**obtain** *t2 e2* **where** *t2-def*[*simp*]: *x* = (*t2*,*e2*) **by** *fastforce*
**show** *?thesis*
**proof**(*cases* (*v*,*e*) ∈ *set* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|})))
**case** *True*
**have** *uneq*: (*t2*,*e2*) ≠ (*t1*,*e1*) **using** *insert.prems(5) t2-def* **by** *fastforce*
**moreover have** *1*: (*t2*,*e2*) ∈ *fset ys* **using** *insert.prems(2)* **by** *fastforce*
**ultimately have** *dlverts t1* ∩ *dlverts t2* = {} **using** *insert.prems(4) wf-lverts* **by** *fastforce*
**then have** *2*: ∀ *x1*∈*fst* ‘ *set* (*dtree-to-list* (*Node r* {|(*t2*, *e2*)|})). *set x1* ∩ *dlverts t1* = {}
**using** *dtree-to-list-x1-disjoint* **by** *fast*
**have** (*darcs t1* ∪ {*e1*}) ∩ (*darcs t2* ∪ {*e2*}) = {}
**using** *insert.prems(4) uneq 1 disjoint-darcs-if-wf-aux5 wf-arcs* **by** *fast*
**then have** *3*: ∀ *e*∈*snd* ‘ *set* (*dtree-to-list* (*Node r* {|(*t2*, *e2*)|})). *e* ∉ *darcs t1* ∪ {*e1*}
**using** *dtree-to-list-e-disjoint* **by** *fast*
**have** [] ∉ *dverts t2* **using** *1 wf-lverts empty-notin-wf-dlverts* **by** *auto*
**then have** ∀ *x1*∈*fst* ‘ *set* (*dtree-to-list* (*Node r* {|(*t2*, *e2*)|})). *x1* ≠ []
**using** *1 dtree-to-list-x-in-dverts* **by** *metis*
**then show** *?thesis* **using** *True 2 3* **by** *fastforce*
**next**
**case** *False*
**have** *xs* |⊆| *finsert x xs* **by** *blast*
**then have** *f-xs*: *ffold ?f z xs* = *ffold ?f' z xs*
**using** *merge-ffold-supset 0* **by** *presburger*
**have** *ffold ?f z* (*finsert x xs*) = *?f x* (*ffold ?f z xs*)
**using** *Comm.ffold-finsert*[*OF insert.hyps*] **by** *blast*
**then have** *0*: *ffold ?f z* (*finsert x xs*) = *?f x* (*ffold ?f' z xs*) **using** *f-xs* **by** *argo*
**then have** *?f x* (*ffold ?f' z xs*) ≠ *ffold ?f' z xs* **using** *not-right* **by** *argo*
**then have** *?f* (*t2*,*e2*) (*ffold ?f' z xs*)
= *?merge cmp'* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|})) (*ffold ?f' z xs*)
**using** *merge-f-merge-if-not-snd t2-def* **by** *blast*
**then have** *ffold ?f z* (*finsert x xs*)
= *?merge cmp'* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|})) (*ffold ?f' z xs*)
**using** *0 t2-def* **by** *argo*
**then have** (*v*,*e*) ∈ *set* (*?merge cmp'* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|})) (*ffold ?f' z xs*))
**using** *insert.prems(6)* **by** *argo*

**then have** $(v,e) \in$ *set* (*ffold* *?f'* *z xs*) **using** *set-merge False* **by** *fast*
**then show** *?thesis* **using** *insert.IH insert.prems* **by** *force*
**qed**
**qed**
**qed** (*auto simp*: *ffold.rep-eq*)

**lemma** *merge-ffold-empty-inter-preserv'*:
⟦*list-dtree* (*Node r* (*finsert x xs*));
$\forall (v,e) \in$ *set z*. *set v* $\cap$ *dlverts t1* $= \{\} \wedge v \neq [] \wedge e \notin$ *darcs t1* $\cup \{e1\}$;
(*t1*,*e1*) $\in$ *fset* (*finsert x xs*); (*t1*,*e1*) $\notin$ *fset xs*; $(v,e) \in$ *set* (*ffold* (*merge-f r xs*) *z xs*)⟧
$\Longrightarrow$ *set v* $\cap$ *dlverts t1* $= \{\} \wedge v \neq [] \wedge e \notin$ *darcs t1* $\cup \{e1\}$
**using** *merge-ffold-empty-inter-preserv*[*of r finsert x xs xs z t1 e1 v e*] **by** *fast*

**lemma** *merge-ffold-set-sub-union*:
*list-dtree* (*Node r xs*)
$\Longrightarrow$ *set* (*ffold* (*merge-f r xs*) [] *xs*) $\subseteq$ ($\bigcup x \in$ *fset xs*. *set* (*dtree-to-list* (*Node r* {|*x*|})))
**proof**(*induction xs*)
**case** (*insert x xs*)
**obtain** *t1 e1* **where** *t1-def*[*simp*]: $x = (t1,e1)$ **by** *fastforce*
**let** *?f* = *merge-f r* (*finsert x xs*)
**let** *?f'* = *merge-f r xs*
**have** (*t1*, *e1*) $\in$ *fset* (*finsert x xs*) **by** *simp*
**moreover have** (*t1*, *e1*) $\notin$ *fset xs* **using** *insert.hyps* **by** *fastforce*
**ultimately have** *xs-val*:
$(\forall (v,e) \in$ *set* (*ffold* *?f'* [] *xs*). *set v* $\cap$ *dlverts t1* $= \{\} \wedge v \neq [] \wedge e \notin$ *darcs t1* $\cup \{e1\}$)
**using** *merge-ffold-empty-inter-preserv'*[*OF insert.prems empty-list-valid-merge*] **by** *blast*
**have** *0*: *list-dtree* (*Node r xs*) **using** *list-dtree-subset insert.prems* **by** *blast*
**have** *ffold ?f* [] (*finsert x xs*) = *?f x* (*ffold ?f* [] *xs*)
**using** *Comm.ffold-finsert*[*OF insert.hyps*] **by** *blast*
**also have** ... = *?f x* (*ffold* *?f'* [] *xs*)
**using** *merge-ffold-supset*[*of xs finsert x xs r* []] *insert.prems* **by** *fastforce*
**finally have** *ffold ?f* [] (*finsert x xs*)
= *Sorting-Algorithms.merge cmp'* (*dtree-to-list* (*Node r* {|*x*|})) (*ffold* *?f'* [] *xs*)
**using** *merge-f-merge-if-conds*[*OF insert.prems xs-val*] **by** *simp*
**then have** *set* (*ffold ?f* [] (*finsert x xs*))
= *set* (*Sorting-Algorithms.merge cmp'* (*dtree-to-list* (*Node r* {|*x*|})) (*ffold* *?f'* [] *xs*))
**by** *argo*
**then have** *set* (*ffold ?f* [] (*finsert x xs*))
= (*set* (*dtree-to-list* (*Node r* {|*x*|})) $\cup$ *set* (*ffold* *?f'* [] *xs*)) **using** *set-merge* **by** *fast*
**then show** *?case* **using** *0 insert.IH insert.prems* **by** *auto*
**qed** (*simp add*: *ffold.rep-eq*)

**lemma** *merge-ffold-nempty*:
⟦*list-dtree* (*Node r xs*); *xs* ≠ {||}⟧ ⟹ *ffold* (*merge-f r xs*) [] *xs* ≠ []
**proof**(*induction xs*)
**case** (*insert x xs*)
**define** *f* **where** *f* = *merge-f r* (*finsert x xs*)
**define** *f'* **where** *f'* = *merge-f r xs*
**let** *?merge* = *Sorting-Algorithms.merge cmp'*
**have** *0*: *list-dtree* (*Node r xs*) **using** *list-dtree-subset insert.prems*(*1*) **by** *blast*
**obtain** *t2 e2* **where** *t2-def*[*simp*]: *x* = (*t2*,*e2*) **by** *fastforce*
**have** (*t2*, *e2*) ∈ *fset* (*finsert x xs*) **by** *simp*
**moreover have** (*t2*, *e2*) ∉ *fset xs* **using** *insert.hyps* **by** *fastforce*
**ultimately have** *xs-val*:
(∀ (*v*,*e*) ∈ *set* (*ffold f'* [] *xs*). *set v* ∩ *dlverts t2* = {} ∧ *v* ≠ [] ∧ *e* ∉ *darcs t2* ∪ {*e2*})
**using** *merge-ffold-empty-inter-preserv'*[*OF insert.prems*(*1*) *empty-list-valid-merge*] *f'-def*
**by** *blast*
**have** *ffold f* [] (*finsert x xs*) = *f x* (*ffold f* [] *xs*)
**using** *Comm.ffold-finsert*[*OF insert.hyps*] *f-def* **by** *blast*
**also have** ... = *f x* (*ffold f'* [] *xs*)
**using** *merge-ffold-supset*[*of xs finsert x xs r* []] *insert.prems*(*1*) *f-def f'-def* **by** *fastforce*
**finally have** *ffold f* [] (*finsert x xs*) = *?merge* (*dtree-to-list* (*Node r* {|*x*|})) (*ffold f'* [] *xs*)
**using** *xs-val insert.prems f-def* **by** *simp*
**then have** *merge*: *ffold f* [] (*finsert x xs*)
= *?merge* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|})) (*ffold f'*[] *xs*)
**using** *t2-def* **by** *blast*
**then show** *?case*
**using** *input-empty-if-merge-empty*[*of cmp' dtree-to-list* (*Node r* {|(*t2*,*e2*)|})] *f-def* **by** *auto*
**qed**(*simp*)

**lemma** *merge-f-ndisjoint-sets-aux*:
¬*disjoint-sets xs*
⟹ ¬((*t*,*e*) ∈ *fset xs* ∧ *disjoint-sets xs* ∧ (∀ (*v*,-) ∈ *set b*. *set v* ∩ *dlverts t* = {} ∧ *v* ≠ []))
**by** *blast*

**lemma** *merge-f-not-list-dtree*: ¬*list-dtree* (*Node r xs*) ⟹ (*merge-f r xs*) *a b* = *b*
**using** *merge-f-alt* **by** *simp*

**lemma** *merge-ffold-empty-if-nwf*: ¬*list-dtree* (*Node r ys*) ⟹ *ffold* (*merge-f r ys*) [] *xs* = []
**proof**(*induction xs*)
**case** (*insert x xs*)
**define** *f* **where** *f* = *merge-f r ys*
**let** *?f* = *merge-f r ys*
**let** *?merge* = *Sorting-Algorithms.merge cmp'*

```
  obtain t2 e2 where t2-def[simp]: x = (t2,e2) by fastforce
  have ffold f [] (finsert x xs) = ?f x (ffold f [] xs)
    using Comm.ffold-finsert[OF insert.hyps] f-def by blast
  then have ffold f [] (finsert x xs) = ffold f [] xs
    using insert.prems merge-f-not-list-dtree by force
  then show ?case using insert f-def by argo
qed (simp add: ffold.rep-eq)

lemma merge-empty-if-nwf: ¬list-dtree (Node r xs) ⟹ merge (Node r xs) = Node
r {||}
  unfolding merge-def using merge-ffold-empty-if-nwf by simp

lemma merge-empty-if-nwf-sucs: ¬list-dtree t1 ⟹ merge t1 = Node (root t1) {||}
  using merge-empty-if-nwf[of root t1 sucs t1] by simp

lemma merge-empty: merge (Node r {||}) = Node r {||}
proof −
  have comp-fun-commute (λ(t, e) b. b)
    by (simp add: comp-fun-commute-const cond-case-prod-eta)
  hence dtree-from-list r (ffold (λ(t, e) b. b) [] {||}) = Node r {||}
    using comp-fun-commute.ffold-empty
    by (smt (verit, best) dtree-from-list.simps(1))
  thus ?thesis
    unfolding merge-def by simp
qed

lemma merge-empty-sucs:
  assumes sucs t1 = {||}
  shows merge t1 = Node (root t1) {||}
proof −
  have dtree-from-list (dtree.root t1) (ffold (λ(t, e) b. b) [] {||}) = Node (dtree.root
t1) {||}
    by (simp add: ffold.rep-eq)
  with assms show ?thesis
    unfolding merge-def by simp
qed

lemma merge-singleton-sucs:
  assumes list-dtree (Node (root t1) (sucs t1)) and sucs t1 ≠ {||}
  shows ∃ t e. merge t1 = Node (root t1) {|(t,e)|}
 unfolding merge-def using merge-ffold-nempty[OF assms] dtree-from-list-singleton
by fast

lemma merge-singleton:
  assumes list-dtree (Node r xs) and xs ≠ {||}
  shows ∃ t e. merge (Node r xs) = Node r {|(t,e)|}
 unfolding merge-def dtree.sel(1) using merge-ffold-nempty[OF assms] dtree-from-list-singleton
  by fastforce
```

**lemma** *merge-cases*: $\exists\, t\ e.\ merge\ (Node\ r\ xs) = Node\ r\ \{|(t,e)|\} \vee merge\ (Node\ r\ xs) = Node\ r\ \{||\}$
**using** *merge-singleton merge-empty-if-nwf merge-empty* **by** *blast*

**lemma** *merge-cases-sucs*:
$\exists\, t\ e.\ merge\ t1 = Node\ (root\ t1)\ \{|(t,e)|\} \vee merge\ t1 = Node\ (root\ t1)\ \{||\}$
**using** *merge-singleton-sucs*[*of t1*] *merge-empty-if-nwf-sucs merge-empty-sucs* **by** *auto*

**lemma** *merge-single-root*:
$(t2,e2) \in fset\ (sucs\ (merge\ (Node\ r\ xs))) \Longrightarrow merge\ (Node\ r\ xs) = Node\ r\ \{|(t2,e2)|\}$
**using** *merge-cases*[*of r xs*] **by** *fastforce*

**lemma** *merge-single-root-sucs*:
$(t2,e2) \in fset\ (sucs\ (merge\ t1)) \Longrightarrow merge\ t1 = Node\ (root\ t1)\ \{|(t2,e2)|\}$
**using** *merge-cases-sucs*[*of t1*] **by** *auto*

**lemma** *merge-single-root1*:
$t2 \in fst\ `\ fset\ (sucs\ (merge\ (Node\ r\ xs))) \Longrightarrow \exists\, e2.\ merge\ (Node\ r\ xs) = Node\ r\ \{|(t2,e2)|\}$
**using** *merge-single-root* **by** *fastforce*

**lemma** *merge-single-root1-sucs*:
$t2 \in fst\ `\ fset\ (sucs\ (merge\ t1)) \Longrightarrow \exists\, e2.\ merge\ t1 = Node\ (root\ t1)\ \{|(t2,e2)|\}$
**using** *merge-single-root-sucs* **by** *fastforce*

**lemma** *merge-nempty-sucs*: $[\![list\text{-}dtree\ t1;\ sucs\ t1 \neq \{||\}]\!] \Longrightarrow sucs\ (merge\ t1) \neq \{||\}$
**using** *merge-singleton-sucs* **by** *fastforce*

**lemma** *merge-nempty*: $[\![list\text{-}dtree\ (Node\ r\ xs);\ xs \neq \{||\}]\!] \Longrightarrow sucs\ (merge\ (Node\ r\ xs)) \neq \{||\}$
**using** *merge-singleton* **by** *fastforce*

**lemma** *merge-xs*: $merge\ (Node\ r\ xs) = dtree\text{-}from\text{-}list\ r\ (ffold\ (merge\text{-}f\ r\ xs)\ []\ xs)$
**unfolding** *merge-def dtree.sel*(*1*) *dtree.sel*(*2*) **by** *blast*

**lemma** *merge-root-eq*[*simp*]: $root\ (merge\ t1) = root\ \ t1$
**unfolding** *merge-def* **by** *simp*

**lemma** *merge-ffold-fsts-in-childverts*:
$[\![list\text{-}dtree\ (Node\ r\ xs);\ y \in fst\ `\ set\ (ffold\ (merge\text{-}f\ r\ xs)\ []\ xs)]\!]$
$\Longrightarrow \exists\, t1 \in fst\ `\ \ fset\ xs.\ y \in dverts\ t1$
**proof**(*induction xs*)
**case** (*insert x xs*)
**obtain** *t1 e1* **where** *t1-def*[*simp*]: $x = (t1,e1)$ **by** *fastforce*
**let** *?f* $= merge\text{-}f\ r\ (finsert\ x\ xs)$
**let** *?f'* $= merge\text{-}f\ r\ xs$

```
  have (t1, e1) ∈ fset (finsert x xs) by simp
  moreover have (t1, e1) ∉ fset xs using insert.hyps by fastforce
  ultimately have xs-val:
    (∀(v,e) ∈ set (ffold ?f' [] xs). set v ∩ dlverts t1 = {} ∧ v ≠ [] ∧ e ∉ darcs t1
∪ {e1})
   using merge-ffold-empty-inter-preserv'[OF insert.prems(1) empty-list-valid-merge]
by blast
  have 0: list-dtree (Node r xs) using list-dtree-subset insert.prems(1) by blast
  then show ?case
  proof(cases y ∈ fst ‘ set (ffold (merge-f r xs) [] xs))
    case True
    then show ?thesis using insert.IH[OF 0] by simp
  next
    case False
    have ffold ?f [] (finsert x xs) = ?f x (ffold ?f [] xs)
      using Comm.ffold-finsert[OF insert.hyps] by blast
    also have ... = ?f x (ffold ?f' [] xs)
      using merge-ffold-supset[of xs finsert x xs r []] insert.prems(1) by fastforce
    finally have ffold ?f [] (finsert x xs)
              = Sorting-Algorithms.merge cmp' (dtree-to-list (Node r {|x|})) (ffold
?f' [] xs)
      using xs-val insert.prems by simp
    then have set (ffold ?f [] (finsert x xs))
         = set (Sorting-Algorithms.merge cmp' (dtree-to-list (Node r {|x|})) (ffold
?f' [] xs))
      by argo
    then have set (ffold ?f [] (finsert x xs))
         = (set (dtree-to-list (Node r {|x|})) ∪ set (ffold ?f' [] xs))
      using set-merge by fast
    then have y ∈ fst ‘ set (dtree-to-list (Node r {|x|})) using False insert.prems
by fast
    then show ?thesis by (simp add: dtree-to-list-x-in-dverts)
  qed
qed (simp add: ffold.rep-eq)

lemma verts-child-if-merge-child:
  assumes t1 ∈ fst ‘ fset (sucs (merge t0)) and x ∈ dverts t1
  shows ∃ t2 ∈ fst ‘ fset (sucs t0). x ∈ dverts t2
proof −
  have 0: list-dtree t0 using assms(1) merge-empty-if-nwf-sucs by fastforce
  have merge t0 ≠ Node (root t0) {||} using assms(1) by force
  then obtain e1 where e1-def: merge t0 = Node (root t0) {|(t1,e1)|}
    using assms(1) merge-single-root1-sucs by blast
  then obtain ys where ys-def:
     (root t1, e1) # ys = ffold (merge-f (root t0) (sucs t0)) [] (sucs t0)
   unfolding merge-def by (metis (no-types, lifting) dtree-to-list.simps(1) dtree-to-from-list-id)
  then have merge t0 = dtree-from-list (root t0) ((root t1, e1) # ys) unfolding
merge-def by simp
  then have t1 = dtree-from-list (root t1) ys using e1-def by simp
```

**then have** *dverts t1* = (*fst* ‘ *set* ((*root t1*, *e1*) # *ys*))
**using** *dtree-from-list-eq-dverts*[*of root t1 ys*] **by** *simp*
**then have** *x* ∈ *fst* ‘ *set* (*ffold* (*merge-f* (*root t0*) (*sucs t0*)) [] (*sucs t0*))
**using** *assms*(*2*) *ys-def* **by** *simp*
**then show** *?thesis* **using** *merge-ffold-fsts-in-childverts*[*of root t0*] *0* **by** *simp*
**qed**

**lemma** *sucs-dverts-eq-dtree-list*:
**assumes** (*t1*,*e1*) ∈ *fset* (*sucs t*) **and** *max-deg t1* ≤ *1*
**shows** *dverts* (*Node* (*root t*) {|(*t1*,*e1*)|}) − {*root t*}
= *fst* ‘ *set* (*dtree-to-list* (*Node* (*root t*) {|(*t1*,*e1*)|}))
**proof** −
**have** {|(*t1*,*e1*)|} |⊆| *sucs t* **using** *assms*(*1*) **by** *fast*
**then have** *wf*: *wf-dverts* (*Node* (*root t*) {|(*t1*,*e1*)|})
**using** *wf-verts wf-dverts-sub* **by** (*metis dtree.exhaust-sel*)
**have** ∀ (*t1*,*e1*) ∈ *fset* (*sucs t*) . *fcard* {|(*t1*,*e1*)|} = *1* **using** *fcard-single-1* **by** *fast*
**moreover have** *max-deg* (*Node* (*root t*) {|(*t1*,*e1*)|}) = *max* (*max-deg t1*) (*fcard* {|(*t1*,*e1*)|})
**using** *mdeg-singleton* **by** *fast*
**ultimately have** *max-deg* (*Node* (*root t*) {|(*t1*,*e1*)|}) ≤ *1*
**using** *assms* **by** *fastforce*
**then show** *?thesis* **using** *dtree-to-list-eq-dverts*[*OF wf*] **by** *simp*
**qed**

**lemma** *merge-ffold-set-eq-union*:
*list-dtree* (*Node r xs*)
⟹ *set* (*ffold* (*merge-f r xs*) [] *xs*) = (⋃ *x*∈*fset xs*. *set* (*dtree-to-list* (*Node r* {|*x*|})))
**proof**(*induction xs*)
**case** (*insert x xs*)
**obtain** *t1 e1* **where** *t1-def*[*simp*]: *x* = (*t1*,*e1*) **by** *fastforce*
**let** *?f* = *merge-f r* (*finsert x xs*)
**let** *?f* ′ = *merge-f r xs*
**have** (*t1*, *e1*) ∈ *fset* (*finsert x xs*) **by** *simp*
**moreover have** (*t1*, *e1*) ∉ *fset xs* **using** *insert.hyps* **by** *fastforce*
**ultimately have** *xs-val*:
(∀ (*v*,*e*) ∈ *set* (*ffold ?f* ′ [] *xs*). *set v* ∩ *dlverts t1* = {} ∧ *v* ≠ [] ∧ *e* ∉ *darcs t1* ∪ {*e1*})
**using** *merge-ffold-empty-inter-preserv*′[*OF insert.prems*(*1*) *empty-list-valid-merge*] **by** *blast*
**have** *1*: *list-dtree* (*Node r xs*) **using** *list-dtree-subset insert.prems*(*1*) **by** *blast*
**have** *ffold ?f* [] (*finsert x xs*) = *?f x* (*ffold ?f* [] *xs*)
**using** *Comm.ffold-finsert*[*OF insert.hyps*] **by** *blast*
**also have** ... = *?f x* (*ffold ?f* ′ [] *xs*)
**using** *merge-ffold-supset*[*of xs finsert x xs r* []] *insert.prems*(*1*) **by** *fastforce*
**finally have** *ffold ?f* [] (*finsert x xs*)
= *Sorting-Algorithms.merge cmp*′ (*dtree-to-list* (*Node r* {|*x*|})) (*ffold ?f* ′ [] *xs*)

```
    using xs-val insert.prems by simp
  then have set (ffold ?f [] (finsert x xs))
      = set (Sorting-Algorithms.merge cmp' (dtree-to-list (Node r {|x|})) (ffold ?f'
[] xs))
    by argo
  then have set (ffold ?f [] (finsert x xs))
       = (set (dtree-to-list (Node r {|x|})) ∪ set (ffold ?f' [] xs)) using set-merge
by fast
  then show ?case using 1 insert.IH by simp
qed (simp add: ffold.rep-eq)

lemma sucs-dverts-no-root:
  (t1,e1) ∈ fset (sucs t) ⟹ dverts (Node (root t) {|(t1,e1)|}) − {root t} = dverts
t1
  using wf-verts wf-dverts'.simps unfolding wf-dverts-iff-dverts' by fastforce

lemma dverts-merge-sub:
  assumes ∀ t ∈ fst ‘ fset (sucs t0). max-deg t ≤ 1
  shows dverts (merge t0) ⊆ dverts t0
proof
  fix x
  assume asm: x ∈ dverts (merge t0)
  show x ∈ dverts t0
  proof(cases x = root (merge t0))
    case True
    then show ?thesis by (simp add: dtree.set-sel(1))
  next
    case False
    then obtain t1 e1 where t1-def: merge t0 = Node (root t0) ({|(t1,e1)|})
      using merge-cases-sucs asm by fastforce
    then have 0: list-dtree (Node (root t0) (sucs t0))
      using merge-empty-if-nwf-sucs by fastforce
    have x ∈ fst ‘ set (ffold (merge-f (root t0) (sucs t0)) [] (sucs t0))
      using t1-def unfolding merge-def using False asm t1-def
       dtree-from-list-eq-dverts[of root t0 ffold (merge-f (root t0) (sucs t0)) [] (sucs
t0)]
      by auto
    then obtain t2 e2 where t2-def:
      (t2,e2) ∈ fset (sucs t0) x ∈ fst ‘ set (dtree-to-list (Node (root t0) {|(t2,e2)|}))
      using merge-ffold-set-sub-union[OF 0] by fast
    then have x ∈ dverts t2 by (simp add: dtree-to-list-x-in-dverts)
    then show ?thesis using t2-def(1) dtree.set-sel(2) by fastforce
  qed
qed

lemma dverts-merge-eq[simp]:
  assumes ∀ t ∈ fst ‘ fset (sucs t). max-deg t ≤ 1
  shows dverts (merge t) = dverts t
proof −
```

**have** $\forall (t1,e1) \in fset\ (sucs\ t).\ dverts\ (Node\ (root\ t)\ \{|(t1,e1)|\}) - \{root\ t\}$
$= fst\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ (root\ t)\ \{|(t1,e1)|\}))$
**using** *sucs-dverts-eq-dtree-list assms*
**by** (*smt* (*verit*, *ccfv-threshold*) *case-prodI2 fst-conv image-iff*)
**then have** $\forall (t1,e1) \in fset\ (sucs\ t).\ dverts\ t1$
$= fst\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ (root\ t)\ \{|(t1,e1)|\}))$
**by** (*metis* (*mono-tags*, *lifting*) *sucs-dverts-no-root case-prodD case-prodI2*)
**then have** $(\bigcup x{\in}fset\ (sucs\ t).\ \bigcup\ (dverts\ `\ Basic\text{-}BNFs.fsts\ x))$
$= (\bigcup x{\in}fset\ (sucs\ t).\ fst\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ (root\ t)\ \{|x|\})))$
**by** *force*
**then have** *dverts t*
$= insert\ (root\ t)\ (\bigcup x{\in}fset\ (sucs\ t).\ fst\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ (root\ t)\ \{|x|\})))$
**using** *dtree.simps*(6)[*of root t sucs t*] **by** *auto*
**also have** $\ldots = insert\ (root\ t)\ (fst\ `\ set\ (ffold\ (merge\text{-}f\ (root\ t)\ (sucs\ t))\ []\ (sucs\ t)))$
**using** *merge-ffold-set-eq-union*[*of root t sucs t*] *list-dtree-axioms* **by** *auto*
**also have** $\ldots = dverts\ (dtree\text{-}from\text{-}list\ (root\ t)\ (ffold\ (merge\text{-}f\ (root\ t)\ (sucs\ t))\ []\ (sucs\ t)))$
**using** *dtree-from-list-eq-dverts*[*of root t*] **by** *blast*
**finally show** *?thesis* **unfolding** *merge-def* **by** *blast*
**qed**

**lemma** *dlverts-merge-eq*[*simp*]:
**assumes** $\forall\, t \in fst\ `\ fset\ (sucs\ t).\ max\text{-}deg\ t \leq 1$
**shows** $dlverts\ (merge\ t) = dlverts\ t$
**using** *dverts-merge-eq*[*OF assms*] **by** (*simp add*: *dlverts-eq-dverts-union*)

**lemma** *sucs-darcs-eq-dtree-list*:
**assumes** $(t1,e1) \in fset\ (sucs\ t)$ **and** $max\text{-}deg\ t1 \leq 1$
**shows** $darcs\ (Node\ (root\ t)\ \{|(t1,e1)|\}) = snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ (root\ t)\ \{|(t1,e1)|\}))$
**proof** −
**have** $\forall (t1,e1) \in fset\ (sucs\ t)\ .\ fcard\ \{|(t1,e1)|\} = 1$ **using** *fcard-single-1* **by** *fast*
**moreover have** $max\text{-}deg\ (Node\ (root\ t)\ \{|(t1,e1)|\}) = max\ (max\text{-}deg\ t1)\ (fcard\ \{|(t1,e1)|\})$
**using** *mdeg-singleton* **by** *fast*
**ultimately have** $max\text{-}deg\ (Node\ (root\ t)\ \{|(t1,e1)|\}) \leq 1$
**using** *assms* **by** *fastforce*
**then show** *?thesis* **using** *dtree-to-list-eq-darcs* **by** *blast*
**qed**

**lemma** *darcs-merge-eq*[*simp*]:
**assumes** $\forall\, t \in fst\ `\ fset\ (sucs\ t).\ max\text{-}deg\ t \leq 1$
**shows** $darcs\ (merge\ t) = darcs\ t$
**proof** −
**have** *0*: *list-dtree* $(Node\ (root\ t)\ (sucs\ t))$ **using** *list-dtree-axioms* **by** *simp*
**have** $\forall (t1,e1) \in fset\ (sucs\ t).\ darcs\ (Node\ (root\ t)\ \{|(t1,e1)|\})$

$= snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ (root\ t)\ \{|(t1,e1)|\}))$
**using** *sucs-darcs-eq-dtree-list assms*
**by** (*smt* (*verit*, *ccfv-threshold*) *case-prodI2 fst-conv image-iff*)
**then have** $\forall (t1,e1) \in fset\ (sucs\ t).\ darcs\ t1 \cup \{e1\}$
$= snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ (root\ t)\ \{|(t1,e1)|\}))$
**by** *simp*
**moreover have** $darcs\ t = (\bigcup (t1,e1) \in fset\ (sucs\ t).\ darcs\ t1 \cup \{e1\})$
**using** *dtree.simps*(7)[*of root t sucs t*] **by** *force*
**ultimately have** $darcs\ t = (\bigcup (t1,e1) \in fset\ (sucs\ t).\ snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ (root\ t)\ \{|(t1,e1)|\})))$
**by** (*smt* (*verit*, *best*) *Sup.SUP-cong case-prodE case-prod-conv*)
**also have** $\ldots = (snd\ `\ set\ (ffold\ (merge\text{-}f\ (root\ t)\ (sucs\ t))\ []\ (sucs\ t)))$
**using** *merge-ffold-set-eq-union*[*OF 0*] **by** *blast*
**also have** $\ldots = darcs\ (dtree\text{-}from\text{-}list\ (root\ t)\ (ffold\ (merge\text{-}f\ (root\ t)\ (sucs\ t))\ []\ (sucs\ t)))$
**using** *dtree-from-list-eq-darcs*[*of root t*] **by** *fast*
**finally show** *?thesis* **unfolding** *merge-def* **by** *blast*
**qed**

### 9.2.4 Merging Preserves Well-Formedness

**lemma** *dtree-to-list-x-in-darcs*:
$x \in snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\})) \Longrightarrow x \in (darcs\ t1 \cup \{e1\})$
**using** *dtree-to-list-sub-darcs* **by** *fastforce*

**lemma** *dtree-to-list-snds-disjoint*:
$(darcs\ t1 \cup \{e1\}) \cap (darcs\ t2 \cup \{e2\}) = \{\}$
$\Longrightarrow snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\})) \cap (darcs\ t2 \cup \{e2\}) = \{\}$
**using** *dtree-to-list-x-in-darcs* **by** *fast*

**lemma** *dtree-to-list-snds-disjoint2*:
$(darcs\ t1 \cup \{e1\}) \cap (darcs\ t2 \cup \{e2\}) = \{\}$
$\Longrightarrow snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\}))$
$\cap\ snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t2,e2)|\})) = \{\}$
**using** *disjoint-iff dtree-to-list-x-in-darcs* **by** *metis*

**lemma** *merge-ffold-arc-inter-preserv*:
$[\![list\text{-}dtree\ (Node\ r\ ys);\ xs\ |\subseteq|\ ys;\ (darcs\ t1 \cup \{e1\}) \cap (snd\ `\ set\ z) = \{\};$
$(t1,e1) \in fset\ ys;\ (t1,e1) \notin fset\ xs]\!]$
$\Longrightarrow (darcs\ t1 \cup \{e1\}) \cap (snd\ `\ set\ (ffold\ (merge\text{-}f\ r\ xs)\ z\ xs)) = \{\}$
**proof**(*induction xs*)
**case** (*insert x xs*)
**let** *?f* $= merge\text{-}f\ r\ (finsert\ x\ xs)$
**let** *?f'* $= merge\text{-}f\ r\ xs$
**let** *?merge* $= Sorting\text{-}Algorithms.merge$
**show** *?case*
**proof**(*cases ffold ?f z* (*finsert x xs*) $= ffold\ ?f'\ z\ xs$)
**case** *True*

```
    then show ?thesis using insert.IH insert.prems by auto
  next
    case False
    obtain t2 e2 where t2-def[simp]: x = (t2,e2) by fastforce
   have 0: list-dtree (Node r (finsert x xs)) using list-dtree-subset insert.prems(1,2)
by blast
    have (t2,e2) ≠ (t1,e1) using insert.prems(5) t2-def by fastforce
    moreover have (t2,e2) ∈ fset ys using insert.prems(2) by fastforce
    moreover have disjoint-darcs ys
     using disjoint-darcs-if-wf[OF list-dtree.wf-arcs [OF insert.prems(1)]] by simp
    ultimately have (darcs t1 ∪ {e1}) ∩ (darcs t2 ∪ {e2}) = {}
      using insert.prems(4) by fast
   then have 1: (darcs t1 ∪ {e1}) ∩ snd ‘ set (dtree-to-list (Node r {|(t2, e2)|}))
= {}
      using dtree-to-list-snds-disjoint by fast
    have 2: (darcs t1 ∪ {e1}) ∩ snd ‘ set (ffold ?f' z xs) = {}
      using insert.IH insert.prems by simp
    have xs |⊆| finsert x xs by blast
    then have f-xs: ffold ?f z xs = ffold ?f' z xs
      using merge-ffold-supset 0 by presburger
    have ffold ?f z (finsert x xs) = ?f x (ffold ?f z xs)
      using Comm.ffold-finsert[OF insert.hyps] by blast
    then have 0: ffold ?f z (finsert x xs) = ?f x (ffold ?f' z xs) using f-xs by argo
    then have ?f x (ffold ?f' z xs) ≠ ffold ?f' z xs using False by argo
    then have ?f (t2,e2) (ffold ?f' z xs)
            = ?merge cmp' (dtree-to-list (Node r {|(t2,e2)|})) (ffold ?f' z xs)
      using merge-f-merge-if-not-snd t2-def by blast
    then have ffold ?f z (finsert x xs)
            = ?merge cmp' (dtree-to-list (Node r {|(t2,e2)|})) (ffold ?f' z xs)
      using 0 t2-def by argo
    then have set (ffold ?f z (finsert x xs))
       = set (dtree-to-list (Node r {|(t2,e2)|})) ∪ set (ffold ?f' z xs)
      using set-merge[of dtree-to-list (Node r {|(t2,e2)|})] by presburger
    then show ?thesis using 1 2 by fast
  qed
qed (auto simp: ffold.rep-eq)

lemma merge-ffold-wf-list-arcs:
  ⟦⋀x. x ∈ fset xs ⟹ wf-darcs (Node r {|x|}); list-dtree (Node r xs)⟧
    ⟹ wf-list-arcs (ffold (merge-f r xs) [] xs)
proof(induction xs)
  case (insert x xs)
  obtain t1 e1 where t1-def[simp]: x = (t1,e1) by fastforce
  let ?f = merge-f r (finsert x xs)
  let ?f' = merge-f r xs
  have 0: (t1, e1) ∈ fset (finsert x xs) by simp
  moreover have t1-not-xs: (t1, e1) ∉ fset xs using insert.hyps by fastforce
  ultimately have xs-val:
    (∀(v,e) ∈ set (ffold ?f' [] xs). set v ∩ dlverts t1 = {} ∧ v ≠ [] ∧ e ∉ darcs t1
```

∪ {*e1*})
  **using** *merge-ffold-empty-inter-preserv'*[*OF insert.prems*(*2*) *empty-list-valid-merge*]
**by** *blast*
  **have** *1*: *wf-list-arcs* (*dtree-to-list* (*Node r* {|*x*|}))
    **using** *insert.prems*(*1*) *0 t1-def wf-list-arcs-if-wf-darcs* **by** *fast*
  **have** *list-dtree* (*Node r xs*) **using** *list-dtree-subset insert.prems*(*2*) **by** *blast*
  **then have** *2*: *wf-list-arcs* (*ffold ?f'* [] *xs*) **using** *insert.IH insert.prems* **by** *auto*
  **have** *darcs* (*Node r* {|*x*|}) ∩ *snd* ‘ *set* (*ffold ?f'* [] *xs*) = {}
   **using** *merge-ffold-arc-inter-preserv*[*OF insert.prems*(*2*), *of xs t1 e1* []] *t1-not-xs*
**by** *auto*
  **then have** *3*: *snd* ‘ *set* (*dtree-to-list* (*Node r* {|*x*|})) ∩ *snd* ‘ *set* (*ffold ?f'* [] *xs*)
= {}
    **using** *dtree-to-list-sub-darcs* **by** *fast*
  **have** *ffold ?f* [] (*finsert x xs*) = *?f x* (*ffold ?f* [] *xs*)
    **using** *Comm.ffold-finsert*[*OF insert.hyps*] **by** *blast*
  **also have** ... = *?f x* (*ffold ?f'* [] *xs*)
    **using** *merge-ffold-supset*[*of xs finsert x xs r* []] *insert.prems*(*2*) **by** *fastforce*
  **finally have** *ffold ?f* [] (*finsert x xs*)
            = *Sorting-Algorithms.merge cmp'* (*dtree-to-list* (*Node r* {|*x*|})) (*ffold*
*?f'* [] *xs*)
    **using** *xs-val insert.prems* **by** *simp*
  **then show** *?case* **using** *wf-list-arcs-merge*[*OF 1 2 3*] **by** *presburger*
**qed** (*simp add*: *ffold.rep-eq*)

**lemma** *merge-wf-darcs*: *wf-darcs* (*merge t*)
**proof** −
  **have** *wf-list-arcs* (*ffold* (*merge-f* (*root t*) (*sucs t*)) [] (*sucs t*))
    **using** *merge-ffold-wf-list-arcs*[*OF wf-darcs-sucs*[*OF wf-arcs*]] *list-dtree-axioms*
**by** *simp*
  **then show** *?thesis* **using** *wf-darcs-iff-wf-list-arcs merge-def* **by** *fastforce*
**qed**

**lemma** *merge-ffold-wf-list-lverts*:
  ⟦⋀*x*. *x* ∈ *fset xs* ⟹ *wf-dlverts* (*Node r* {|*x*|}); *list-dtree* (*Node r xs*)⟧
   ⟹ *wf-list-lverts* (*ffold* (*merge-f r xs*) [] *xs*)
**proof**(*induction xs*)
  **case** (*insert x xs*)
  **obtain** *t1 e1* **where** *t1-def*[*simp*]: *x* = (*t1*,*e1*) **by** *fastforce*
  **let** *?f* = *merge-f r* (*finsert x xs*)
  **let** *?f'* = *merge-f r xs*
  **have** *0*: (*t1*, *e1*) ∈ *fset* (*finsert x xs*) **by** *simp*
  **moreover have** (*t1*, *e1*) ∉ *fset xs* **using** *insert.hyps* **by** *fastforce*
  **ultimately have** *xs-val*:
    (∀(*v*,*e*) ∈ *set* (*ffold ?f'* [] *xs*). *set v* ∩ *dlverts t1* = {} ∧ *v* ≠ [] ∧ *e* ∉ *darcs t1*
∪ {*e1*})
   **using** *merge-ffold-empty-inter-preserv'*[*OF insert.prems*(*2*) *empty-list-valid-merge*]
**by** *blast*
  **have** *1*: *wf-list-lverts* (*dtree-to-list* (*Node r* {|*x*|}))
    **using** *insert.prems*(*1*) *0 t1-def wf-list-lverts-if-wf-dlverts* **by** *fast*

```
  have list-dtree (Node r xs) using list-dtree-subset insert.prems(2) by blast
  then have 2: wf-list-lverts (ffold ?f' [] xs) using insert.IH insert.prems by auto
  have ∀ v2∈fst ‘ set (ffold ?f' [] xs). set v2 ∩ dlverts t1 = {}
    using xs-val by fastforce
  then have 3: ∀ v1∈fst ‘ set (dtree-to-list (Node r {|x|})). ∀ v2∈fst ‘ set (ffold ?f'
[] xs).
                  set v1 ∩ set v2 = {}
    using dtree-to-list-x1-list-disjoint t1-def by fast
  have ffold ?f [] (finsert x xs) = ?f x (ffold ?f [] xs)
    using Comm.ffold-finsert[OF insert.hyps] by blast
  also have ... = ?f x (ffold ?f' [] xs)
    using merge-ffold-supset[of xs finsert x xs r []] insert.prems(2) by fastforce
  finally have ffold ?f [] (finsert x xs)
             = Sorting-Algorithms.merge cmp' (dtree-to-list (Node r {|x|})) (ffold
?f' [] xs)
    using xs-val insert.prems by simp
  then show ?case using wf-list-lverts-merge[OF 1 2 3] by presburger
qed (simp add: ffold.rep-eq)

lemma merge-ffold-root-inter-preserv:
  ⟦list-dtree (Node r xs); ∀ t1 ∈ fst ‘ fset xs. set r' ∩ dlverts t1 = {};
    ∀ v1 ∈ fst ‘ set z. set r' ∩ set v1 = {}; (v,e) ∈ set (ffold (merge-f r xs) z xs)⟧
    ⟹ set r' ∩ set v = {}
proof(induction xs)
  case (insert x xs)
  let ?f = merge-f r (finsert x xs)
  let ?f' = merge-f r xs
  let ?merge = Sorting-Algorithms.merge
  have 0: list-dtree (Node r xs) using insert.prems(1) list-dtree-subset by blast
  show ?case
  proof(cases ffold ?f z (finsert x xs) = ffold ?f' z xs)
    case True
    then show ?thesis using insert.IH[OF 0] insert.prems(2−4) by simp
  next
    case not-right: False
    obtain t2 e2 where t2-def[simp]: x = (t2,e2) by fastforce
    show ?thesis
    proof(cases (v,e) ∈ set (dtree-to-list (Node r {|(t2,e2)|})))
      case True
        then show ?thesis using dtree-to-list-x1-list-disjoint insert.prems(2) by
fastforce
    next
      case False
      have xs |⊆| finsert x xs by blast
      then have f-xs: ffold ?f z xs = ffold ?f' z xs
        using merge-ffold-supset[of xs finsert x xs] insert.prems(1) by blast
      have ffold ?f z (finsert x xs) = ?f x (ffold ?f z xs)
        using Comm.ffold-finsert[OF insert.hyps] by blast
      then have 1: ffold ?f z (finsert x xs) = ?f x (ffold ?f' z xs) using f-xs by
```

```
argo
    then have ?f x (ffold ?f' z xs) ≠ ffold ?f' z xs using not-right by argo
    then have ?f (t2,e2) (ffold ?f' z xs)
          = ?merge cmp' (dtree-to-list (Node r {|(t2,e2)|})) (ffold ?f' z xs)
      using merge-f-merge-if-not-snd t2-def by blast
    then have ffold ?f z (finsert x xs)
          = ?merge cmp' (dtree-to-list (Node r {|(t2,e2)|})) (ffold ?f' z xs)
      using 1 t2-def by argo
   then have (v,e) ∈ set (?merge cmp' (dtree-to-list (Node r {|(t2,e2)|})) (ffold
?f' z xs))
      using insert.prems(4) by argo
    then have (v,e) ∈ set (ffold ?f' z xs) using set-merge False by fast
    then show ?thesis using insert.IH insert.prems(2−3) 0 by auto
  qed
 qed
qed (fastforce simp: ffold.rep-eq)

lemma merge-wf-dlverts: wf-dlverts (merge t)
proof −
 have 0: list-dtree (Node (root t) (sucs t)) using list-dtree-axioms by simp
 have 1: ∀ t1∈fst ‘ fset (sucs t). set (root t) ∩ dlverts t1 = {}
  using wf-lverts wf-dlverts.simps[of root t] by fastforce
 have ∀ v∈fst ‘ set (ffold (merge-f (root t) (sucs t)) [] (sucs t)). set (root t) ∩ set
v = {}
  using wf-lverts merge-ffold-root-inter-preserv[OF 0 1] by force
 moreover have wf-list-lverts (ffold (merge-f (root t) (sucs t)) [] (sucs t))
  using merge-ffold-wf-list-lverts[OF wf-dlverts-sucs[OF wf-lverts] 0] by simp
 moreover have root t ≠ [] using wf-lverts wf-dlverts.elims(2) by fastforce
 ultimately show ?thesis unfolding merge-def using wf-dlverts-iff-wf-list-lverts
by blast
qed

theorem merge-list-dtree: list-dtree (merge t)
 using merge-wf-dlverts merge-wf-darcs list-dtree-def by blast

corollary merge-ranked-dtree: ranked-dtree (merge t) cmp
 using merge-list-dtree ranked-dtree-def ranked-dtree-axioms by auto
```

### 9.2.5 Additional Merging Properties

```
lemma merge-ffold-distinct:
 ⟦list-dtree (Node r xs); ∀ t1 ∈ fst ‘ fset xs. ∀ v∈dverts t1. distinct v;
   ∀ v1 ∈ fst ‘ set z. distinct v1; v ∈ fst ‘ set (ffold (merge-f r xs) z xs)⟧
   ⟹ distinct v
proof(induction xs)
 case (insert x xs)
 let ?f = merge-f r (finsert x xs)
 let ?f' = merge-f r xs
 let ?merge = Sorting-Algorithms.merge
```

```
  have 0: list-dtree (Node r xs) using insert.prems(1) list-dtree-subset by blast
  show ?case
  proof(cases ffold ?f z (finsert x xs) = ffold ?f' z xs)
    case True
    then show ?thesis using insert.IH[OF 0] insert.prems(2−4) by simp
  next
    case not-right: False
    obtain t2 e2 where t2-def[simp]: x = (t2,e2) by fastforce
    show ?thesis
    proof(cases v ∈ fst ‘ set (dtree-to-list (Node r {|(t2,e2)|})))
      case True
      have ∀ v∈dverts t2. distinct v using insert.prems(2) by simp
      then have 2: ∀ v∈fst ‘ set (dtree-to-list (Node r {|(t2,e2)|})). distinct v
        by (simp add: dtree-to-list-x-in-dverts)
      then show ?thesis using True by auto
    next
      case False
      have xs |⊆| finsert x xs by blast
      then have f-xs: ffold ?f z xs = ffold ?f' z xs
        using merge-ffold-supset insert.prems(1) by presburger
      have ffold ?f z (finsert x xs) = ?f x (ffold ?f z xs)
        using Comm.ffold-finsert[OF insert.hyps] by blast
      then have 1: ffold ?f z (finsert x xs) = ?f x (ffold ?f' z xs) using f-xs by
argo
      then have ?f x (ffold ?f' z xs) ≠ ffold ?f' z xs using not-right by argo
      then have ?f (t2,e2) (ffold ?f' z xs)
              = ?merge cmp' (dtree-to-list (Node r {|(t2,e2)|})) (ffold ?f' z xs)
        using merge-f-merge-if-not-snd t2-def by blast
      then have ffold ?f z (finsert x xs)
              = ?merge cmp' (dtree-to-list (Node r {|(t2,e2)|})) (ffold ?f' z xs)
        using 1 t2-def by argo
     then have v ∈ fst ‘ set (?merge cmp' (dtree-to-list (Node r {|(t2,e2)|})) (ffold
?f' z xs))
        using insert.prems(4) by argo
      then have v ∈ fst ‘ set (ffold ?f' z xs) using set-merge False by fast
      then show ?thesis using insert.IH[OF 0] insert.prems(2−3) by simp
    qed
  qed
qed (fastforce simp: ffold.rep-eq)

lemma distinct-merge:
  assumes ∀ v∈dverts t. distinct v and v∈dverts (merge t)
  shows distinct v
proof(cases v = root t)
  case True
  then show ?thesis by (simp add: dtree.set-sel(1) assms(1))
next
  case False
  then have 0: v ∈ fst ‘ set (ffold (merge-f (root t) (sucs t)) [] (sucs t))
```

```
    using merge-def assms(2) dtree-from-list-eq-dverts[of root t] by auto
  moreover have ∀ t1∈fst ‘ fset (sucs t). ∀ v∈dverts t1. distinct v
    using assms(1) dverts-child-subset[of root t sucs t] by auto
  moreover have ∀ v1∈fst ‘ set []. distinct v1 by simp
  moreover have 0: list-dtree (Node (root t) (sucs t)) using list-dtree-axioms by
simp
  ultimately show ?thesis using merge-ffold-distinct by fast
qed

lemma merge-hd-root-eq[simp]: hd (root (merge t1)) = hd (root t1)
  unfolding merge-def by auto

lemma merge-ffold-hd-is-child:
  ⟦list-dtree (Node r xs); xs ≠ {||}⟧
    ⟹ ∃(t1,e1) ∈ fset xs. hd (ffold (merge-f r xs) [] xs) = (root t1,e1)
proof(induction xs)
  case (insert x xs)
 interpret Comm: comp-fun-commute merge-f r (finsert x xs) by (rule merge-commute)
  define f where f = merge-f r (finsert x xs)
  define f' where f' = merge-f r xs
  let ?merge = Sorting-Algorithms.merge cmp'
  have 0: list-dtree (Node r xs) using list-dtree-subset insert.prems(1) by blast
  obtain t2 e2 where t2-def[simp]: x = (t2,e2) by fastforce
  have i1: ∃(t1, e1)∈fset (finsert x xs). hd (dtree-to-list (Node r {|(t2,e2)|})) =
(root t1, e1)
    by simp
  have (t2, e2) ∈ fset (finsert x xs) by simp
  moreover have (t2, e2) ∉ fset xs using insert.hyps by fastforce
  ultimately have xs-val:
    (∀(v,e) ∈ set (ffold f' [] xs). set v ∩ dlverts t2 = {} ∧ v ≠ [] ∧ e ∉ darcs t2 ∪
{e2})
   using merge-ffold-empty-inter-preserv'[OF insert.prems(1) empty-list-valid-merge]
f'-def
    by blast
  have ffold f [] (finsert x xs) = f x (ffold f [] xs)
    using Comm.ffold-finsert[OF insert.hyps] f-def by blast
  also have ... = f x (ffold f' [] xs)
    using merge-ffold-supset[of xs finsert x xs r []] insert.prems(1) f-def f'-def by
fastforce
  finally have ffold f [] (finsert x xs) = ?merge (dtree-to-list (Node r {|x|})) (ffold
f' [] xs)
    using xs-val insert.prems f-def by simp
  then have merge: ffold f [] (finsert x xs)
            = ?merge (dtree-to-list (Node r {|(t2,e2)|})) (ffold f'[] xs)
    using t2-def by blast
  show ?case
  proof(cases xs = {||})
    case True
    then show ?thesis using merge i1 f-def by (auto simp: ffold.rep-eq)
```

```
  next
    case False
    then have i2: ∃(t1,e1) ∈ fset (finsert x xs). hd (ffold f' [] xs) = (root t1,e1)
      using insert.IH[OF 0] f'-def by simp
    show ?thesis using merge-hd-exists-preserv[OF i1 i2] merge f-def by simp
  qed
qed(simp)

lemma merge-ffold-nempty-if-child:
  assumes (t1,e1) ∈ fset (sucs (merge t0))
  shows ffold (merge-f (root t0) (sucs t0)) [] (sucs t0) ≠ []
  using assms unfolding merge-def by auto

lemma merge-ffold-hd-eq-child:
  assumes (t1,e1) ∈ fset (sucs (merge t0))
  shows hd (ffold (merge-f (root t0) (sucs t0)) [] (sucs t0)) = (root t1,e1)
proof −
  have merge t0 = (dtree-from-list (root t0) (ffold (merge-f (root t0) (sucs t0)) []
(sucs t0)))
    unfolding merge-def by blast
  have merge t0 = (Node (root t0) {|(t1,e1)|}) using merge-cases-sucs[of t0]
assms by auto
  have 0: (Node (root t0) {|(t1,e1)|})
     = (dtree-from-list (root t0) (ffold (merge-f (root t0) (sucs t0)) [] (sucs t0)))
    using merge-cases-sucs[of t0] assms unfolding merge-def by fastforce
  then obtain ys where (root t1, e1) # ys = ffold (merge-f (root t0) (sucs t0))
[] (sucs t0)
    using dtree-from-list-eq-singleton[OF 0] by blast
  then show ?thesis using list.sel(1)[of (root t1, e1) ys] by simp
qed

lemma merge-child-in-orig:
  assumes (t1,e1) ∈ fset (sucs (merge t0))
  shows ∃(t2,e2) ∈ fset (sucs t0). (root t2,e2) = (root t1,e1)
proof −
  have 0: list-dtree (Node (root t0) (sucs t0)) using assms merge-empty-if-nwf-sucs
by fastforce
  have sucs t0 ≠ {||} using assms merge-empty-sucs by fastforce
  then obtain t2 e2 where t2-def: (t2,e2) ∈ fset (sucs t0)
    hd (ffold (merge-f (root t0) (sucs t0)) [] (sucs t0)) = (root t2,e2)
    using merge-ffold-hd-is-child[OF 0] by blast
  then show ?thesis using merge-ffold-hd-eq-child[OF assms] by auto
qed

lemma ffold-singleton: comp-fun-commute f ⟹ ffold f z {|x|} = f x z
  using comp-fun-commute.ffold-finsert
  by (metis comp-fun-commute.ffold-empty finsert-absorb finsert-not-fempty)

lemma ffold-singleton1:
```

⟦*comp-fun-commute* (λ*a b*. *if P a b then Q a b else R a b*); *P x z*⟧
⟹ *ffold* (λ*a b*. *if P a b then Q a b else R a b*) *z* {|*x*|} = *Q x z*
**using** *ffold-singleton* **by** *fastforce*

**lemma** *ffold-singleton2*:
⟦*comp-fun-commute* (λ*a b*. *if P a b then Q a b else R a b*); ¬*P x z*⟧
⟹ *ffold* (λ*a b*. *if P a b then Q a b else R a b*) *z* {|*x*|} = *R x z*
**using** *ffold-singleton* **by** *fastforce*

**lemma** *merge-ffold-singleton-if-wf*:
**assumes** *list-dtree* (*Node r* {|(*t1*,*e1*)|})
**shows** *ffold* (*merge-f r* {|(*t1*,*e1*)|}) [] {|(*t1*,*e1*)|} = *dtree-to-list* (*Node r* {|(*t1*,*e1*)|})
**proof** −
**interpret** *Comm*: *comp-fun-commute merge-f r* {|(*t1*,*e1*)|} **by** (*rule merge-commute*)
**define** *f* **where** *f* = *merge-f r* {|(*t1*,*e1*)|}
**have** *ffold f* [] {|(*t1*,*e1*)|} = *f* (*t1*,*e1*) (*ffold f* [] {||})
**using** *Comm.ffold-finsert f-def* **by** *blast*
**then show** *?thesis* **using** *f-def assms* **by** (*simp add*: *ffold.rep-eq*)
**qed**

**lemma** *merge-singleton-if-wf*:
**assumes** *list-dtree* (*Node r* {|(*t1*,*e1*)|})
**shows** *merge* (*Node r* {|(*t1*,*e1*)|}) = *dtree-from-list r* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|}))
**using** *merge-ffold-singleton-if-wf*[*OF assms*] *merge-xs* **by** *simp*

**lemma** *merge-disjoint-if-child*:
*merge* (*Node r* {|(*t1*,*e1*)|}) = *Node r* {|(*t2*,*e2*)|} ⟹ *list-dtree* (*Node r* {|(*t1*,*e1*)|})
**using** *merge-empty-if-nwf* **by** *fastforce*

**lemma** *merge-root-child-eq*:
*merge* (*Node r* {|(*t1*,*e1*)|}) = *Node r* {|(*t2*,*e2*)|} ⟹ *root t1* = *root t2*
**using** *merge-singleton-if-wf*[*OF merge-disjoint-if-child*] **by** *fastforce*

**lemma** *merge-ffold-split-subtree*:
⟦∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*; *list-dtree* (*Node r xs*);
*as*@(*v*,*e*)#*bs* = *ffold* (*merge-f r xs*) [] *xs*⟧
⟹ ∃ *ys*. *strict-subtree* (*Node v ys*) (*Node r xs*) ∧ *dverts* (*Node v ys*) ⊆ (*fst* ‘ *set* ((*v*,*e*)#*bs*))
**proof**(*induction xs arbitrary*: *as bs*)
**case** (*insert x xs*)
**obtain** *t1 e1* **where** *t1-def*[*simp*]: *x* = (*t1*,*e1*) **by** *fastforce*
**define** *f′* **where** *f′* = *merge-f r xs*
**let** *?f* = *merge-f r* (*finsert x xs*)
**let** *?f′* = *merge-f r xs*
**have** (*t1*, *e1*) ∈ *fset* (*finsert x xs*) **by** *simp*
**moreover have** (*t1*, *e1*) ∉ *fset xs* **using** *insert.hyps* **by** *fastforce*
**ultimately have** *xs-val*:
(∀ (*v*,*e*) ∈ *set* (*ffold ?f′* [] *xs*). *set v* ∩ *dlverts t1* = {} ∧ *v* ≠ [] ∧ *e* ∉ *darcs t1*

$\cup$ {*e1*})
  **using** *merge-ffold-empty-inter-preserv'*[*OF insert.prems*(*2*) *empty-list-valid-merge*]
**by** *blast*
  **have** *0*: $\forall$ *t* $\in$ *fst* ‘ *fset xs*. *max-deg t* $\leq$ *1* **using** *insert.prems*(*1*) **by** *simp*
  **have** *1*: *list-dtree* (*Node r xs*) **using** *list-dtree-subset insert.prems*(*2*) **by** *blast*
  **have** *ffold ?f* [] (*finsert x xs*) = *?f x* (*ffold ?f* [] *xs*)
    **using** *Comm.ffold-finsert*[*OF insert.hyps*] **by** *blast*
  **also have** ... = *?f x* (*ffold ?f'* [] *xs*)
    **using** *merge-ffold-supset*[*of xs finsert x xs r* []] *insert.prems*(*2*) **by** *fastforce*
  **finally have** *ind*: *ffold ?f* [] (*finsert x xs*)
          = *Sorting-Algorithms.merge cmp'* (*dtree-to-list* (*Node r* {|*x*|})) (*ffold f'*
[] *xs*)
    **using** *insert.prems*(*2*) *xs-val f'-def* **by** *simp*
  **have** *max-deg* (*fst x*) $\leq$ *1* **using** *insert.prems*(*1*) **by** *simp*
  **then have** *max-deg* (*Node r* {|*x*|}) $\leq$ *1*
    **using** *mdeg-child-sucs-eq-if-gt1*[*of r fst x snd x root* (*fst x*)] **by** *fastforce*
  **then have** $\forall$ *as bs*. *as*@(*v*,*e*)#*bs* = *dtree-to-list* (*Node r* {|*x*|}) $\longrightarrow$
      ($\exists$ *zs*. *strict-subtree* (*Node v zs*) (*Node r* {|*x*|})
        $\wedge$ *dverts* (*Node v zs*) $\subseteq$ *fst* ‘ *set* ((*v*,*e*)#*bs*))
    **using** *dtree-to-list-split-subtree-dverts-eq-fsts'* **by** *fast*
  **then have** *left*: $\forall$ *as bs*. *as*@(*v*,*e*)#*bs* = *dtree-to-list* (*Node r* {|*x*|}) $\longrightarrow$
      ($\exists$ *zs*. *strict-subtree* (*Node v zs*) (*Node r* (*finsert x xs*))
        $\wedge$ *dverts* (*Node v zs*) $\subseteq$ *fst* ‘ *set* ((*v*,*e*)#*bs*))
    **using** *strict-subtree-singleton*[**where** *xs*=*finsert x xs*] **by** *blast*
  **have** $\forall$ *as bs*. *as*@(*v*,*e*)#*bs* = *ffold f'* [] *xs* $\longrightarrow$
      ($\exists$ *zs*. *strict-subtree* (*Node v zs*) (*Node r xs*)
        $\wedge$ *dverts* (*Node v zs*) $\subseteq$ *fst* ‘ *set* ((*v*,*e*)#*bs*))
    **using** *insert.IH*[*OF 0 1*] *f'-def* **by** *blast*
  **then have** *right*: $\forall$ *as bs*. *as*@(*v*,*e*)#*bs* = *ffold f'* [] *xs* $\longrightarrow$
      ($\exists$ *zs*. *strict-subtree* (*Node v zs*) (*Node r* (*finsert x xs*))
        $\wedge$ *dverts* (*Node v zs*) $\subseteq$ *fst* ‘ *set* ((*v*,*e*)#*bs*))
   **using** *strict-subtree-subset*[**where** *r*=*r* **and** *xs*=*xs* **and** *ys*=*finsert x xs*] **by** *fast*
  **then show** *?case* **using** *merge-split-supset-strict-subtree*[*OF left right*] *ind in-*
*sert.prems*(*3*) **by** *simp*
**qed** (*simp add*: *ffold.rep-eq*)

**lemma** *merge-strict-subtree-dverts-sup*:
  **assumes** $\forall$ *t* $\in$ *fst* ‘ *fset* (*sucs t*). *max-deg t* $\leq$ *1*
      **and** *strict-subtree* (*Node r xs*) (*merge t*)
    **shows** $\exists$ *ys*. *is-subtree* (*Node r ys*) *t* $\wedge$ *dverts* (*Node r ys*) $\subseteq$ *dverts* (*Node r xs*)
**proof** −
  **have** *0*: *list-dtree* (*Node* (*root t*) (*sucs t*)) **using** *list-dtree-axioms* **by** *simp*
  **have** $\forall$ *as r e bs*. *as*@(*r*,*e*)#*bs* = *ffold* (*merge-f* (*root t*) (*sucs t*)) [] (*sucs t*)
    $\longrightarrow$ ($\exists$ *ys*. *strict-subtree* (*Node r ys*) (*Node* (*root t*) (*sucs t*))
        $\wedge$ *dverts* (*Node r ys*) $\subseteq$ *fst* ‘ *set* ((*r*,*e*)#*bs*))
    **using** *merge-ffold-split-subtree*[*OF assms*(*1*) *0*] **by** *blast*
  **then have** $\forall$ *as r e bs*. *as*@(*r*,*e*)#*bs* = *ffold* (*merge-f* (*root t*) (*sucs t*)) [] (*sucs*
*t*) $\longrightarrow$
    ($\exists$ *ys*. *strict-subtree* (*Node r ys*) *t* $\wedge$ *dverts* (*Node r ys*) $\subseteq$ *fst* ‘ *set* ((*r*,*e*)#*bs*))

**by** *simp*
**obtain** *as e bs* **where** *bs-def*: *as@(r,e)#bs = ffold (merge-f (root t) (sucs t)) [] (sucs t)*
**using** *assms(2) dtree-from-list-uneq-sequence-xs[of r]* **unfolding** *merge-def* **by** *blast*
**have** *wf-dverts (merge t)* **by** (*simp add*: *merge-wf-dlverts wf-dverts-if-wf-dlverts*)
**then have** *wf*: *wf-dverts (dtree-from-list (root t) (as@(r,e)#bs))*
**unfolding** *merge-def bs-def* **.**
**moreover obtain** *ys* **where**
*strict-subtree (Node r ys) t dverts (Node r ys) ⊆ fst ‘ set ((r,e)#bs)*
**using** *merge-ffold-split-subtree[OF assms(1) 0 bs-def]* **by** *auto*
**moreover have** *strict-subtree (Node r xs) (dtree-from-list (root t) (as@(r,e)#bs))*
**using** *assms(2)* **unfolding** *bs-def merge-def* **.**
**ultimately show** *?thesis*
**using** *dtree-from-list-dverts-subset-wfdverts1* **unfolding** *strict-subtree-def* **by** *fast*
**qed**

**lemma** *merge-subtree-dverts-supset*:
**assumes** ∀ *t∈fst ‘ fset (sucs t). max-deg t ≤ 1* **and** *is-subtree (Node r xs) (merge t)*
**shows** ∃ *ys. is-subtree (Node r ys) t ∧ dverts (Node r ys) ⊆ dverts (Node r xs)*
**proof**(*cases Node r xs = merge t*)
**case** *True*
**then obtain** *ys* **where** *t = Node r ys* **using** *merge-root-eq dtree.exhaust-sel dtree.sel(1)* **by** *metis*
**then show** *?thesis* **using** *dverts-merge-eq[OF assms(1)] True* **by** *auto*
**next**
**case** *False*
**then show** *?thesis* **using** *merge-strict-subtree-dverts-sup assms strict-subtree-def* **by** *blast*
**qed**

**lemma** *merge-subtree-dlverts-supset*:
**assumes** ∀ *t∈fst ‘ fset (sucs t). max-deg t ≤ 1* **and** *is-subtree (Node r xs) (merge t)*
**shows** ∃ *ys. is-subtree (Node r ys) t ∧ dlverts (Node r ys) ⊆ dlverts (Node r xs)*
**proof** −
**obtain** *ys* **where** *is-subtree (Node r ys) t dverts (Node r ys) ⊆ dverts (Node r xs)*
**using** *merge-subtree-dverts-supset[OF assms]* **by** *blast*
**then show** *?thesis* **using** *dlverts-eq-dverts-union[of Node r ys] dlverts-eq-dverts-union* **by** *fast*
**qed**

**end**

## 9.3 Normalizing Dtrees

**context** *ranked-dtree*
**begin**

### 9.3.1 Definitions

**function** *normalize1* :: (*'a list*,*'b*) *dtree* ⇒ (*'a list*,*'b*) *dtree* **where**
*normalize1* (*Node r* {|(*t1*,*e*)|}) =
(*if rank* (*rev* (*root t1*)) < *rank* (*rev r*) *then Node* (*r*@*root t1*) (*sucs t1*)
*else Node r* {|(*normalize1 t1*,*e*)|})
| ∀ *x*. *xs* ≠ {|*x*|} ⟹ *normalize1* (*Node r xs*) = *Node r* ((λ(*t*,*e*). (*normalize1 t*,*e*))
|‘| *xs*)
**by** (*metis darcs-mset.cases old.prod.exhaust*) *fast+*
**termination by** *lexicographic-order*

**lemma** *normalize1-size-decr*[*termination-simp*]:
*normalize1 t1* ≠ *t1* ⟹ *size* (*normalize1 t1*) < *size t1*
**proof**(*induction t1 rule*: *normalize1.induct*)
**case** (*1 r t e*)
**then show** *?case*
**proof**(*cases rank* (*rev* (*root t*)) < *rank* (*rev r*))
**case** *True*
**then show** *?thesis* **using** *dtree-size-eq-root*[*of root t sucs t*] **by** *simp*
**next**
**case** *False*
**then show** *?thesis* **using** *dtree-size-img-le 1* **by** *auto*
**qed**
**next**
**case** (*2 xs r*)
**then have** *0*: ∀ *t* ∈ *fst* ‘ *fset xs*. *size* (*normalize1 t*) ≤ *size t* **by** *fastforce*
**moreover have** ∃ *t* ∈ *fst* ‘ *fset xs*. *size* (*normalize1 t*) < *size t*
**using** *elem-neq-if-fset-neq*[*of normalize1 xs*] *2* **by** *fastforce*
**ultimately show** *?case* **using** *dtree-size-img-lt 2.hyps* **by** *auto*
**qed**

**lemma** *normalize1-size-le*: *size* (*normalize1 t1*) ≤ *size t1*
**by**(*cases normalize1 t1*=*t1*) (*auto dest*: *normalize1-size-decr*)

**fun** *normalize* :: (*'a list*,*'b*) *dtree* ⇒ (*'a list*,*'b*) *dtree* **where**
*normalize t1* = (*let t2* = *normalize1 t1 in if t1* = *t2 then t2 else normalize t2*)

### 9.3.2 Basic Proofs

**lemma** *root-normalize1-eq1*:
¬*rank* (*rev* (*root t1*)) < *rank* (*rev r*) ⟹ *root* (*normalize1* (*Node r* {|(*t1*,*e1*)|}))
= *r*
**by** *simp*

**lemma** *root-normalize1-eq1'*:

$\neg rank\ (rev\ (root\ t1)) \leq rank\ (rev\ r) \Longrightarrow root\ (normalize1\ (Node\ r\ \{|(t1,e1)|\})) = r$
**by** *simp*

**lemma** *root-normalize1-eq2*: $\forall x.\ xs \neq \{|x|\} \Longrightarrow root\ (normalize1\ (Node\ r\ xs)) = r$
**by** *simp*

**lemma** *fset-img-eq*: $\forall x \in fset\ xs.\ f\ x = x \Longrightarrow f\ |`|\ xs = xs$
**using** *fset-inject*[*of xs f* $|`|$ *xs*] **by** *simp*

**lemma** *fset-img-uneq*: $f\ |`|\ xs \neq xs \Longrightarrow \exists x \in fset\ xs.\ f\ x \neq x$
**using** *fset-img-eq* **by** *fastforce*

**lemma** *fset-img-uneq-prod*: $(\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs \neq xs \Longrightarrow \exists (t,e) \in fset\ xs.\ f\ t \neq t$
**using** *fset-img-uneq*[*of* $\lambda(t,e).\ (f\ t,\ e)$ *xs*] **by** *auto*

**lemma** *contr-if-normalize1-uneq*:
*normalize1 t1* $\neq$ *t1*
$\Longrightarrow \exists v\ t2\ e2.\ is\text{-}subtree\ (Node\ v\ \{|(t2,e2)|\})\ t1 \wedge rank\ (rev\ (root\ t2)) < rank\ (rev\ v)$
**proof**(*induction t1 rule*: *normalize1.induct*)
**case** (*2 xs r*)
**then show** *?case* **using** *fset-img-uneq-prod*[*of normalize1 xs*] **by** *fastforce*
**qed**(*fastforce*)

**lemma** *contr-before-normalize1*:
⟦*is-subtree* (*Node v* $\{|(t1,e1)|\}$) (*normalize1 t3*); *rank* (*rev* (*root t1*)) < *rank* (*rev v*)⟧
$\Longrightarrow \exists v'\ t2\ e2.\ is\text{-}subtree\ (Node\ v'\ \{|(t2,e2)|\})\ t3 \wedge rank\ (rev\ (root\ t2)) < rank\ (rev\ v')$
**using** *contr-if-normalize1-uneq* **by** *force*

### 9.3.3 Normalizing Preserves Well-Formedness

**lemma** *normalize1-darcs-sub*: *darcs* (*normalize1 t1*) ⊆ *darcs t1*
**proof**(*induction t1 rule*: *normalize1.induct*)
**case** (*1 r t e*)
**then show** *?case*
**proof**(*cases rank* (*rev* (*root t*)) < *rank* (*rev r*))
**case** *True*
**then have** *darcs* (*normalize1* (*Node r* $\{|(t,e)|\}$)) = *darcs* (*Node* (*r@root t*) (*sucs t*)) **by** *simp*
**also have** ... = *darcs* (*Node* (*root t*) (*sucs t*)) **using** *darcs-sub-if-children-sub* **by** *fast*
**finally show** *?thesis* **by** *auto*
**next**
**case** *False*

```
    then show ?thesis using 1 by auto
  qed
qed (fastforce)

lemma disjoint-darcs-normalize1:
  wf-darcs t1 ⟹ disjoint-darcs ((λ(t,e). (normalize1 t,e)) |`| (sucs t1))
  using disjoint-darcs-img[OF disjoint-darcs-if-wf, of t1 normalize1]
  by (simp add: normalize1-darcs-sub)

lemma wf-darcs-normalize1: wf-darcs t1 ⟹ wf-darcs (normalize1 t1)
proof(induction t1 rule: normalize1.induct)
  case (1 r t e)
  show ?case
  proof(cases rank (rev (root t)) < rank (rev r))
    case True
    then show ?thesis
      using 1.prems dtree.collapse singletonI finsert.rep-eq case-prodD
      unfolding wf-darcs-iff-darcs'
    by (metis (no-types, lifting) wf-darcs'.simps bot-fset.rep-eq normalize1.simps(1))
  next
    case False
    have disjoint-darcs {|(normalize1 t,e)|}
      using normalize1-darcs-sub disjoint-darcs-if-wf-xs[OF 1.prems] by auto
    then show ?thesis using 1 False unfolding wf-darcs-iff-darcs' by force
  qed
next
  case (2 xs r)
  then show ?case
    using disjoint-darcs-normalize1[OF 2.prems]
    by (fastforce simp: wf-darcs-iff-darcs')
qed

lemma normalize1-dlverts-eq[simp]: dlverts (normalize1 t1) = dlverts t1
proof(induction t1 rule: normalize1.induct)
  case (1 r t e)
  then show ?case
  proof(cases rank (rev (root t)) < rank (rev r))
    case True
    then show ?thesis using dlverts.simps[of root t sucs t] by force
  next
    case False
    then show ?thesis using 1 by auto
  qed
qed (fastforce)

lemma normalize1-dverts-contr-subtree:
  ⟦v ∈ dverts (normalize1 t1); v ∉ dverts t1⟧
    ⟹ ∃ v2 t2 e2. is-subtree (Node v2 {|(t2,e2)|}) t1
        ∧ v2 @ root t2 = v ∧ rank (rev (root t2)) < rank (rev v2)
```

**proof**(*induction t1 rule*: *normalize1.induct*)
**case** (*1 r t e*)
**show** *?case*
**proof**(*cases rank* (*rev* (*root t*)) < *rank* (*rev r*))
**case** *True*
**then show** *?thesis* **using** *1.prems dverts-suc-subseteq* **by** *fastforce*
**next**
**case** *False*
**then show** *?thesis* **using** *1* **by** *auto*
**qed**
**qed**(*fastforce*)

**lemma** *normalize1-dverts-app-contr*:
⟦*v* ∈ *dverts* (*normalize1 t1*); *v* ∉ *dverts t1*⟧
⟹ ∃ *v1*∈*dverts t1*. ∃ *v2*∈*dverts t1*. *v1* @ *v2* = *v* ∧ *rank* (*rev v2*) < *rank* (*rev v1*)
**using** *normalize1-dverts-contr-subtree*
**by** (*fastforce simp*: *single-subtree-root-dverts single-subtree-child-root-dverts*)

**lemma** *disjoint-dlverts-img*:
**assumes** *disjoint-dlverts xs* **and** ∀ (*t*,*e*) ∈ *fset xs*. *dlverts* (*f t*) ⊆ *dlverts t*
**shows** *disjoint-dlverts* ((λ(*t*,*e*). (*f t*,*e*)) |‘| *xs*) (**is** *disjoint-dlverts ?xs*)
**proof** (*rule ccontr*)
**assume** ¬ *disjoint-dlverts ?xs*
**then obtain** *x1 e1 y1 e2* **where** *asm*: (*x1*,*e1*) ∈ *fset ?xs* (*y1*,*e2*) ∈ *fset ?xs*
*dlverts x1* ∩ *dlverts y1* ≠ {} ∧ (*x1*,*e1*)≠(*y1*,*e2*) **by** *blast*
**then obtain** *x2* **where** *x2-def*: *f x2* = *x1* (*x2*,*e1*) ∈ *fset xs* **by** *auto*
**obtain** *y2* **where** *y2-def*: *f y2* = *y1* (*y2*,*e2*) ∈ *fset xs* **using** *asm*(*2*) **by** *auto*
**have** *dlverts x1* ⊆ *dlverts x2* **using** *assms*(*2*) *x2-def* **by** *fast*
**moreover have** *dlverts y1* ⊆ *dlverts y2* **using** *assms*(*2*) *y2-def* **by** *fast*
**ultimately have** ¬ *disjoint-dlverts xs* **using** *asm*(*3*) *x2-def y2-def* **by** *blast*
**then show** *False* **using** *assms*(*1*) **by** *blast*
**qed**

**lemma** *disjoint-dlverts-normalize1*:
*disjoint-dlverts xs* ⟹ *disjoint-dlverts* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs*)
**using** *disjoint-dlverts-img*[*of xs*] **by** *simp*

**lemma** *disjoint-dlverts-normalize1-sucs*:
*disjoint-dlverts* (*sucs t1*) ⟹ *disjoint-dlverts* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| (*sucs t1*))
**using** *disjoint-dlverts-img*[*of sucs t1*] **by** *simp*

**lemma** *disjoint-dlverts-normalize1-wf*:
*wf-dlverts t1* ⟹ *disjoint-dlverts* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| (*sucs t1*))
**using** *disjoint-dlverts-img*[*OF disjoint-dlverts-if-wf*, *of t1*] **by** *simp*

**lemma** *disjoint-dlverts-normalize1-wf'*:
*wf-dlverts* (*Node r xs*) ⟹ *disjoint-dlverts* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs*)

**using** *disjoint-dlverts-img*[*OF disjoint-dlverts-if-wf*, *of Node r xs*] **by** *simp*

**lemma** *root-empty-inter-dlverts-normalize1*:
**assumes** *wf-dlverts t1* **and** (*x1*,*e1*) ∈ *fset* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |`| (*sucs t1*))
**shows** *set* (*root t1*) ∩ *dlverts x1* = {}
**proof** (*rule ccontr*)
**assume** *asm*: *set* (*root t1*) ∩ *dlverts x1* ≠ {}
**obtain** *x2* **where** *x2-def*: *normalize1 x2* = *x1* (*x2*,*e1*) ∈ *fset* (*sucs t1*) **using** *assms*(*2*) **by** *auto*
**have** *set* (*root t1*) ∩ *dlverts x2* ≠ {} **using** *x2-def*(*1*) *asm* **by** *force*
**then show** *False* **using** *x2-def*(*2*) *assms*(*1*) *wf-dlverts.simps*[*of root t1 sucs t1*]
**by** *auto*
**qed**

**lemma** *wf-dlverts-normalize1*: *wf-dlverts t1* ⟹ *wf-dlverts* (*normalize1 t1*)
**proof**(*induction t1 rule*: *normalize1.induct*)
**case** (*1 r t e*)
**show** *?case*
**proof**(*cases rank* (*rev* (*root t*)) < *rank* (*rev r*))
**case** *True*
**have** *0*: ∀ (*t1*,*e1*) ∈ *fset* (*sucs t*). *wf-dlverts t1*
**using** *1.prems wf-dlverts.simps*[*of root t sucs t*] **by** *auto*
**have** ∀ (*t1*,*e1*) ∈ *fset* (*sucs t*). *set* (*root t*) ∩ *dlverts t1* = {}
**using** *1.prems wf-dlverts.simps*[*of root t*] **by** *fastforce*
**then have** ∀ (*t1*,*e1*) ∈ *fset* (*sucs t*). *set* (*r*@*root t*) ∩ *dlverts t1* = {}
**using** *suc-in-dlverts 1.prems* **by** *fastforce*
**then show** *?thesis* **using** *True 0 disjoint-dlverts-if-wf*[*of t*] *1.prems* **by** *auto*
**next**
**case** *False*
**then show** *?thesis*
**using** *root-empty-inter-dlverts-normalize1*[*OF 1.prems*] *disjoint-dlverts-normalize1 1* **by** *auto*
**qed**
**next**
**case** (*2 xs r*)
**have** ∀ (*t1*,*e1*) ∈ *fset* ((λ(*t*, *e*). (*normalize1 t*, *e*)) |`| *xs*). *set r* ∩ *dlverts t1* = {}
**using** *root-empty-inter-dlverts-normalize1*[*OF 2.prems*] **by** *force*
**then show** *?case* **using** *disjoint-dlverts-normalize1 2* **by** *auto*
**qed**

**corollary** *list-dtree-normalize1*: *list-dtree* (*normalize1 t*)
**using** *wf-dlverts-normalize1*[*OF wf-lverts*] *wf-darcs-normalize1*[*OF wf-arcs*] *list-dtree-def*
**by** *blast*

**corollary** *ranked-dtree-normalize1*: *ranked-dtree* (*normalize1 t*) *cmp*
**using** *list-dtree-normalize1 ranked-dtree-def ranked-dtree-axioms* **by** *blast*

**lemma** *normalize-darcs-sub*: *darcs* (*normalize t1*) ⊆ *darcs t1*

**apply**(*induction t1 rule*: *normalize.induct*)
**by** (*smt* (*verit*) *normalize1-darcs-sub normalize.simps subset-trans*)

**lemma** *normalize-dlverts-eq*: *dlverts* (*normalize t1*) = *dlverts t1*
**by**(*induction t1 rule*: *normalize.induct*) (*metis* (*full-types*) *normalize.elims normalize1-dlverts-eq*)

**theorem** *ranked-dtree-normalize*: *ranked-dtree* (*normalize t*) *cmp*
**using** *ranked-dtree-axioms* **apply**(*induction t rule*: *normalize.induct*)
**by** (*smt* (*verit*) *ranked-dtree.normalize.elims ranked-dtree.ranked-dtree-normalize1*)

### 9.3.4 Distinctness and hd preserved

**lemma** *distinct-normalize1*: ⟦∀ *v*∈*dverts t*. *distinct v*; *v*∈*dverts* (*normalize1 t*)⟧ ⟹ *distinct v*
**using** *ranked-dtree-axioms* **proof**(*induction t rule*: *normalize1.induct*)
**case** (*1 r t e*)
**then interpret** *R*: *ranked-dtree Node r* {|(*t*, *e*)|} *rank* **by** *blast*
**show** *?case*
**proof**(*cases rank* (*rev* (*root t*)) < *rank* (*rev r*))
**case** *True*
**interpret** *T*: *ranked-dtree t rank* **using** *R.ranked-dtree-rec* **by** *auto*
**have** *set r* ∩ *set* (*root t*) = {}
**using** *R.wf-lverts dlverts.simps*[*of root t sucs t*] **by** *auto*
**then have** *distinct* (*r*@*root t*) **by** (*auto simp*: *dtree.set-sel*(*1*) *1.prems*(*1*))
**moreover have** ∀ *v* ∈ (⋃(*t*, *e*)∈*fset* (*sucs t*). *dverts t*). *distinct v*
**using** *1.prems*(*1*) *dtree.set*(*1*)[*of root t sucs t*] **by** *fastforce*
**ultimately show** *?thesis* **using** *dverts-root-or-child 1.prems*(*2*) *True* **by** *auto*
**next**
**case** *False*
**then show** *?thesis* **using** *R.ranked-dtree-rec 1* **by** *auto*
**qed**
**next**
**case** (*2 xs r*)
**then interpret** *R*: *ranked-dtree Node r xs rank* **by** *blast*
**show** *?case* **using** *R.ranked-dtree-rec 2* **by** *fastforce*
**qed**

**lemma** *distinct-normalize*: ∀ *v*∈*dverts t*. *distinct v* ⟹ ∀ *v*∈*dverts* (*normalize t*). *distinct v*
**using** *ranked-dtree-axioms* **proof**(*induction t rule*: *normalize.induct*)
**case** (*1 t*)
**then interpret** *T1*: *ranked-dtree t rank* **by** *blast*
**interpret** *T2*: *ranked-dtree normalize1 t rank* **by** (*simp add*: *T1.ranked-dtree-normalize1*)
**show** *?case*
**by** (*smt* (*verit*, *del-insts*) *1 T1.distinct-normalize1 T2.ranked-dtree-axioms normalize.simps*)
**qed**

**lemma** *normalize1-hd-root-eq*[*simp*]:
**assumes** *root t1* ≠ []
**shows** *hd* (*root* (*normalize1 t1*)) = *hd* (*root t1*)
**proof**(*cases* ∀ *x*. *sucs t1* ≠ {|*x*|})
**case** *True*
**then show** *?thesis* **using** *normalize1.simps*(*2*)[*of sucs t1 root t1*] **by** *simp*
**next**
**case** *False*
**then obtain** *t e* **where** {|(*t*, *e*)|} = *sucs t1* **by** *auto*
**then show** *?thesis* **using** *normalize1.simps*(*1*)[*of root t1 t e*] *assms* **by** *simp*
**qed**

**corollary** *normalize1-hd-root-eq'*:
*wf-dlverts t1* ⟹ *hd* (*root* (*normalize1 t1*)) = *hd* (*root t1*)
**using** *normalize1-hd-root-eq*[*of t1*] *wf-dlverts.simps*[*of root t1 sucs t1*] **by** *simp*

**lemma** *normalize1-root-nempty*:
**assumes** *root t1* ≠ []
**shows** *root* (*normalize1 t1*) ≠ []
**proof**(*cases* ∀ *x*. *sucs t1* ≠ {|*x*|})
**case** *True*
**then show** *?thesis* **using** *normalize1.simps*(*2*)[*of sucs t1 root t1*] *assms* **by** *simp*
**next**
**case** *False*
**then obtain** *t e* **where** {|(*t*, *e*)|} = *sucs t1* **by** *auto*
**then show** *?thesis* **using** *normalize1.simps*(*1*)[*of root t1 t e*] *assms* **by** *simp*
**qed**

**lemma** *normalize-hd-root-eq*[*simp*]: *root t1* ≠ [] ⟹ *hd* (*root* (*normalize t1*)) = *hd* (*root t1*)
**using** *ranked-dtree-axioms* **proof**(*induction t1 rule*: *normalize.induct*)
**case** (*1 t*)
**then show** *?case*
**proof**(*cases t* = *normalize1 t*)
**case** *False*
**then have** *normalize t* = *normalize* (*normalize1 t*) **by** (*simp add*: *Let-def*)
**then show** *?thesis* **using** *1 normalize1-root-nempty* **by** *force*
**qed**(*simp*)
**qed**

**corollary** *normalize-hd-root-eq'*[*simp*]: *wf-dlverts t1* ⟹ *hd* (*root* (*normalize t1*)) = *hd* (*root t1*)
**using** *normalize-hd-root-eq wf-dlverts.simps*[*of root t1 sucs t1*] **by** *simp*

### 9.3.5 Normalize and Sorting

**lemma** *normalize1-uneq-if-contr*:
⟦*is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2*; *rank* (*rev* (*root t1*)) < *rank* (*rev r1*); *wf-darcs t2*⟧

$\Longrightarrow$ *t2* $\neq$ *normalize1 t2*
**proof**(*induction t2 rule*: *normalize1.induct*)
**case** (*1 r t e*)
**then show** *?case*
**proof**(*cases rank* (*rev* (*root t*)) < *rank* (*rev r*))
**case** *True*
**then show** *?thesis* **using** *combine-uneq* **by** *fastforce*
**next**
**case** *False*
**then show** *?thesis* **using** *1* **by** *auto*
**qed**
**next**
**case** (*2 xs r*)
**then obtain** *t e* **where** *t-def*: (*t*,*e*) $\in$ *fset xs is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t*
**by** *auto*
**then have** *t* $\neq$ *normalize1 t* **using** *2* **by** *fastforce*
**then have** (*normalize1 t*, *e*) $\notin$ *fset xs*
**using** *2.prems*(*3*) *t-def*(*1*) **by** (*auto simp*: *wf-darcs-iff-darcs′*)
**moreover have** (*normalize1 t*, *e*) $\in$ *fset* ((*λ*(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs*)
**using** *t-def*(*1*) **by** *auto*
**ultimately have** (*λ*(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs* $\neq$ *xs* **using** *t-def*(*1*) **by** *fastforce*
**then show** *?case* **using** *2.hyps* **by** *simp*
**qed**

**lemma** *sorted-ranks-if-normalize1-eq*:
⟦*wf-darcs t2*; *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2*; *t2* = *normalize1 t2*⟧
$\Longrightarrow$ *rank* (*rev r1*) $\leq$ *rank* (*rev* (*root t1*))
**using** *normalize1-uneq-if-contr* **by** *fastforce*

**lemma** *normalize-sorted-ranks*:
⟦*is-subtree* (*Node r* {|(*t1*,*e1*)|}) (*normalize t*)⟧ $\Longrightarrow$ *rank* (*rev r*) $\leq$ *rank* (*rev* (*root t1*))
**using** *ranked-dtree-axioms* **proof**(*induction t rule*: *normalize.induct*)
**case** (*1 t*)
**then interpret** *T*: *ranked-dtree t* **by** *blast*
**show** *?case*
**using** *1 sorted-ranks-if-normalize1-eq*[*OF T.wf-arcs*]
**by** (*smt* (*verit*, *ccfv-SIG*) *T.ranked-dtree-normalize1 normalize.simps*)
**qed**

**lift-definition** *cmp″* :: (*′a list*×*′b*) *comparator* **is**
(*λx y*. *if rank* (*rev* (*fst x*)) < *rank* (*rev* (*fst y*)) *then Less*
*else if rank* (*rev* (*fst x*)) > *rank* (*rev* (*fst y*)) *then Greater*
*else Equiv*)
**by** (*simp add*: *comparator-def*)

**lemma** *dtree-to-list-sorted-if-no-contr*:
⟦$\bigwedge$*r1 t1 e1*. *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2* $\Longrightarrow$ *rank* (*rev r1*) $\leq$ *rank* (*rev* (*root t1*))⟧

$\Longrightarrow$ *sorted cmp''* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|}))
**proof**(*induction cmp'' dtree-to-list* (*Node r* {|(*t2*,*e2*)|}) *arbitrary*: *r t2 e2 rule*: *sorted.induct*)
**case** (*2 x*)
**then show** *?case* **using** *sorted-single*[*of cmp'' x*] **by** *simp*
**next**
**case** (*3 y x xs*)
**then obtain** *r1 t1 e1* **where** *r1-def*: *t2* = *Node r1* {|(*t1*,*e1*)|}
**using** *dtree-to-list.elims*[*of t2*] **by** *fastforce*
**have** *y* = (*root t2*,*e2*) **using** *3.hyps*(*2*) *r1-def* **by** *simp*
**moreover have** *x* = (*root t1*,*e1*) **using** *3.hyps*(*2*) *r1-def* **by** *simp*
**moreover have** *rank* (*rev* (*root t2*)) ≤ *rank* (*rev* (*root t1*)) **using** *3.prems r1-def* **by** *auto*
**ultimately have** *compare cmp'' y x* ≠ *Greater* **using** *cmp''.rep-eq* **by** *simp*
**moreover have** *sorted cmp''* (*dtree-to-list t2*) **using** *3 r1-def* **by** *auto*
**ultimately show** *?case* **using** *3 r1-def* **by** *simp*
**qed**(*simp*)

**lemma** *dtree-to-list-sorted-if-no-contr'*:
⟦⋀*r1 t1 e1*. *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2* $\Longrightarrow$ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))⟧
$\Longrightarrow$ *sorted cmp''* (*dtree-to-list t2*)
**using** *dtree-to-list-sorted-if-no-contr*[*of t2*] *sorted-Cons-imp-sorted* **by** *fastforce*

**lemma** *dtree-to-list-sorted-if-subtree*:
⟦*is-subtree t1 t2*;
⋀*r1 t1 e1*. *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2* $\Longrightarrow$ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))⟧
$\Longrightarrow$ *sorted cmp''* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|}))
**using** *dtree-to-list-sorted-if-no-contr subtree-trans* **by** *blast*

**lemma** *dtree-to-list-sorted-if-subtree'*:
⟦*is-subtree t1 t2*;
⋀*r1 t1 e1*. *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2* $\Longrightarrow$ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))⟧
$\Longrightarrow$ *sorted cmp''* (*dtree-to-list t1*)
**using** *dtree-to-list-sorted-if-no-contr' subtree-trans* **by** *blast*

**lemma** *normalize-dtree-to-list-sorted*:
*is-subtree t1* (*normalize t*) $\Longrightarrow$ *sorted cmp''* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|}))
**using** *dtree-to-list-sorted-if-subtree normalize-sorted-ranks* **by** *blast*

**lemma** *normalize-dtree-to-list-sorted'*:
*is-subtree t1* (*normalize t*) $\Longrightarrow$ *sorted cmp''* (*dtree-to-list t1*)
**using** *dtree-to-list-sorted-if-subtree' normalize-sorted-ranks* **by** *blast*

**lemma** *gt-if-rank-contr*: *rank* (*rev r0*) < *rank* (*rev r*) $\Longrightarrow$ *compare cmp''* (*r*, *e*) (*r0*, *e0*) = *Greater*
**by** (*auto simp*: *cmp''.rep-eq*)

**lemma** *rank-le-if-ngt*: *compare cmp'' (r, e) (r0, e0) ≠ Greater ⟹ rank (rev r) ≤ rank (rev r0)*
**using** *gt-if-rank-contr* **by** *force*

**lemma** *rank-le-if-sorted-from-list*:
**assumes** *sorted cmp'' ((v1,e1)#ys)* **and** *is-subtree (Node r0 {|(t0,e0)|}) (dtree-from-list v1 ys)*
**shows** *rank (rev r0) ≤ rank (rev (root t0))*
**proof** −
**obtain** *e as bs* **where** *e-def*: *as @ (r0, e) # (root t0, e0) # bs = ((v1,e1)#ys)*
**using** *dtree-from-list-sequence*[*OF assms*(*2*)] **by** *blast*
**then have** *sorted cmp'' (as @ (r0, e) # (root t0, e0) # bs)* **using** *assms*(*1*) **by** *simp*
**then have** *sorted cmp'' ((r0, e) # (root t0, e0) # bs)* **using** *sorted-app-r* **by** *blast*
**then show** *?thesis* **using** *rank-le-if-ngt* **by** *auto*
**qed**

**lemma** *cmp'-gt-if-cmp''-gt*: *compare cmp'' x y = Greater ⟹ compare cmp' x y = Greater*
**by** (*auto simp*: *cmp'.rep-eq cmp''.rep-eq split*: *if-splits*)

**lemma** *cmp'-lt-if-cmp''-lt*: *compare cmp'' x y = Less ⟹ compare cmp' x y = Less*
**by** (*auto simp*: *cmp'.rep-eq cmp''.rep-eq*)

**lemma** *cmp''-ge-if-cmp'-gt*:
*compare cmp' x y = Greater ⟹ compare cmp'' x y = Greater ∨ compare cmp'' x y = Equiv*
**by** (*auto simp*: *cmp'.rep-eq cmp''.rep-eq split*: *if-splits*)

**lemma** *cmp''-nlt-if-cmp'-gt*: *compare cmp' x y = Greater ⟹ compare cmp'' y x ≠ Greater*
**by** (*auto simp*: *cmp'.rep-eq cmp''.rep-eq*)

**interpretation** *Comm*: *comp-fun-commute merge-f r xs* **by** (*rule merge-commute*)

**lemma** *sorted-cmp''-merge*:
⟦*sorted cmp'' xs*; *sorted cmp'' ys*⟧ ⟹ *sorted cmp'' (Sorting-Algorithms.merge cmp' xs ys)*
**proof**(*induction xs ys taking*: *cmp' rule*: *Sorting-Algorithms.merge.induct*)
**case** (*3 x xs y ys*)
**let** *?merge = Sorting-Algorithms.merge cmp'*
**show** *?case*
**proof**(*cases compare cmp' x y = Greater*)
**case** *True*
**have** *?merge (x # xs) (y#ys) = y # (?merge (x # xs) ys)* **using** *True* **by** *simp*
**moreover have** *sorted cmp'' (?merge (x # xs) ys)* **using** *3 True sorted-Cons-imp-sorted*

**by** *fast*
  **ultimately show** *?thesis*
   **using** *cmp''-nlt-if-cmp'-gt*[*OF True*] *3.prems sorted-rec*[*of cmp'' y*]
    *merge.elims*[*of cmp' x#xs ys ?merge* (*x # xs*) *ys*]
   **by** *metis*
 **next**
  **case** *False*
  **have** *?merge* (*x#xs*) (*y#ys*) = *x #* (*?merge xs* (*y#ys*)) **using** *False* **by** *simp*
 **moreover have** *sorted cmp''* (*?merge xs* (*y#ys*)) **using** *3 False sorted-Cons-imp-sorted*
**by** *fast*
  **ultimately show** *?thesis*
   **using** *cmp'-gt-if-cmp''-gt False 3.prems sorted-rec*[*of cmp'' x*]
    *merge.elims*[*of cmp' xs y#ys ?merge xs* (*y#ys*)]
   **by** *metis*
 **qed**
**qed**(*auto*)

**lemma** *merge-ffold-sorted*:
 ⟦*list-dtree* (*Node r xs*); ⋀*t2 r1 t1 e1*. ⟦*t2* ∈ *fst* ‘ *fset xs*; *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2*⟧
  ⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))⟧
  ⟹ *sorted cmp''* (*ffold* (*merge-f r xs*) [] *xs*)
**proof**(*induction xs*)
 **case** (*insert x xs*)
 **interpret** *Comm*: *comp-fun-commute merge-f r* (*finsert x xs*) **by** (*rule merge-commute*)
 **define** *f* **where** *f* = *merge-f r* (*finsert x xs*)
 **define** *f'* **where** *f'* = *merge-f r xs*
 **let** *?merge* = *Sorting-Algorithms.merge cmp'*
 **have** *0*: *list-dtree* (*Node r xs*) **using** *list-dtree-subset insert.prems*(*1*) **by** *blast*
 **obtain** *t2 e2* **where** *t2-def*[*simp*]: *x* = (*t2*,*e2*) **by** *fastforce*
 **have** *ind1*: *sorted cmp''* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|}))
  **using** *dtree-to-list-sorted-if-no-contr insert.prems*(*2*) **by** *fastforce*
 **have** ⋀*t2 r1 t1 e1*. ⟦*t2* ∈ *fst* ‘ *fset xs*; *is-subtree* (*Node r1* {|(*t1*, *e1*)|}) *t2*⟧
    ⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
  **using** *insert.prems*(*2*) **by** *fastforce*
 **then have** *ind2*: *sorted cmp''* (*ffold f'* [] *xs*) **using** *insert.IH*[*OF 0*] *f'-def* **by** *blast*
 **have** (*t2*, *e2*) ∈ *fset* (*finsert x xs*) **by** *simp*
 **moreover have** (*t2*, *e2*) ∉ *fset xs* **using** *insert.hyps* **by** *fastforce*
 **ultimately have** *xs-val*:
  (∀(*v*,*e*) ∈ *set* (*ffold f'* [] *xs*). *set v* ∩ *dlverts t2* = {} ∧ *v* ≠ [] ∧ *e* ∉ *darcs t2* ∪ {*e2*})
  **using** *merge-ffold-empty-inter-preserv'*[*OF insert.prems*(*1*) *empty-list-valid-merge*] *f'-def*
  **by** *blast*
 **have** *ffold f* [] (*finsert x xs*) = *f x* (*ffold f* [] *xs*)
  **using** *Comm.ffold-finsert*[*OF insert.hyps*] *f-def* **by** *blast*
 **also have** ... = *f x* (*ffold f'* [] *xs*)
  **using** *merge-ffold-supset*[*of xs finsert x xs r* []] *insert.prems*(*1*) *f-def f'-def* **by**

*fastforce*
**finally have** *ffold f* [] (*finsert x xs*) = *?merge* (*dtree-to-list* (*Node r* {|*x*|})) (*ffold f′* [] *xs*)
**using** *xs-val insert.prems f-def* **by** *simp*
**then have** *merge*: *ffold f* [] (*finsert x xs*)
= *?merge* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|})) (*ffold f′*[] *xs*)
**using** *t2-def* **by** *blast*
**then show** *?case* **using** *sorted-cmp′′-merge*[*OF ind1 ind2*] *f-def* **by** *auto*
**qed** (*simp add*: *ffold.rep-eq*)

**lemma** *not-single-subtree-if-nwf*:
¬*list-dtree* (*Node r xs*) ⟹ ¬*is-subtree* (*Node r1* {|(*t1*,*e1*)|}) (*merge* (*Node r xs*))
**using** *merge-empty-if-nwf* **by** *simp*

**lemma** *not-single-subtree-if-nwf-sucs*:
¬*list-dtree t2* ⟹ ¬*is-subtree* (*Node r1* {|(*t1*,*e1*)|}) (*merge t2*)
**using** *merge-empty-if-nwf-sucs* **by** *simp*

**lemma** *merge-strict-subtree-nocontr*:
**assumes** ⋀*t2 r1 t1 e1*. ⟦*t2* ∈ *fst* ‘ *fset xs*; *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2*⟧
⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
**and** *strict-subtree* (*Node r1* {|(*t1*,*e1*)|}) (*merge* (*Node r xs*))
**shows** *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
**proof**(*cases list-dtree* (*Node r xs*))
**case** *True*
**obtain** *e as bs* **where** *e-def*: *as* @ (*r1*, *e*) # (*root t1*, *e1*) # *bs* = *ffold* (*merge-f r xs*) [] *xs*
**using** *dtree-from-list-uneq-sequence assms*(*2*) **unfolding** *merge-def dtree.sel strict-subtree-def*
**by** *fast*
**have** *sorted cmp′′* (*ffold* (*merge-f r xs*) [] *xs*)
**using** *merge-ffold-sorted*[*OF True assms*(*1*)] **by** *simp*
**then have** *sorted cmp′′* ((*r1*, *e*) # (*root t1*, *e1*) # *bs*)
**using** *e-def sorted-app-r*[*of cmp′′ as* (*r1*, *e*) # (*root t1*, *e1*) # *bs*] **by** *simp*
**then show** *?thesis* **using** *rank-le-if-sorted-from-list* **by** *fastforce*
**next**
**case** *False*
**then show** *?thesis* **using** *not-single-subtree-if-nwf assms*(*2*) **by** (*simp add*: *strict-subtree-def*)
**qed**

**lemma** *merge-strict-subtree-nocontr2*:
**assumes** ⋀*r1 t1 e1*. *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) (*Node r xs*)
⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
**and** *strict-subtree* (*Node r1* {|(*t1*,*e1*)|}) (*merge* (*Node r xs*))
**shows** *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
**using** *merge-strict-subtree-nocontr*[*OF assms*] **by** *fastforce*

**lemma** *merge-strict-subtree-nocontr-sucs*:
**assumes** ⋀*t2 r1 t1 e1*. ⟦*t2* ∈ *fst* ‘ *fset* (*sucs t0*); *is-subtree* (*Node r1* {|(*t1*,*e1*)|})

*t2*⟧
⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
**and** *strict-subtree* (*Node r1* {|(*t1*,*e1*)|}) (*merge t0*)
**shows** *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
**using** *merge-strict-subtree-nocontr*[*of sucs t0 r1 t1 e1 root t0*] *assms* **by** *simp*

**lemma** *merge-strict-subtree-nocontr-sucs2*:
**assumes** ⋀*r1 t1 e1*. *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2* ⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
**and** *strict-subtree* (*Node r1* {|(*t1*,*e1*)|}) (*merge t2*)
**shows** *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
**using** *merge-strict-subtree-nocontr2*[*of root t2 sucs t2*] *assms* **by** *auto*

**lemma** *no-contr-imp-parent*:
⟦*is-subtree* (*Node r1* {|(*t1*,*e1*)|}) (*Node r xs*) ⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*));
*t2* ∈ *fst* ‘ *fset xs*; *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2*⟧
⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
**using** *subtree-if-child subtree-trans* **by** *fast*

**lemma** *no-contr-imp-subtree*:
⟦⋀*t2 r1 t1 e1*. ⟦*t2* ∈ *fst* ‘ *fset xs*; *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2*⟧
⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*));
*is-subtree* (*Node r1* {|(*t1*,*e1*)|}) (*Node r xs*); ∀ *x*. *xs* ≠ {|*x*|}⟧
⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
**by** *fastforce*

**lemma** *no-contr-imp-subtree-fcard*:
⟦⋀*t2 r1 t1 e1*. ⟦*t2* ∈ *fst* ‘ *fset xs*; *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2*⟧
⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*));
*is-subtree* (*Node r1* {|(*t1*,*e1*)|}) (*Node r xs*); *fcard xs* ≠ *1*⟧
⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
**using** *fcard-single-1-iff*[*of xs*] **by** *fastforce*

**end**

## 9.4 Removing Wedges

**context** *ranked-dtree*
**begin**

**fun** *merge1* :: (*'a list*,*'b*) *dtree* ⇒ (*'a list*,*'b*) *dtree* **where**
*merge1* (*Node r xs*) = (
*if fcard xs* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*) *then merge* (*Node r xs*)
*else Node r* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs*))

**lemma** *merge1-dverts-eq*[*simp*]: *dverts* (*merge1 t*) = *dverts t*
**using** *ranked-dtree-axioms* **proof**(*induction t*)
**case** (*Node r xs*)

```
  then interpret R: ranked-dtree Node r xs rank by blast
  show ?case
  proof(cases fcard xs > 1 ∧ (∀ t ∈ fst ‘ fset xs. max-deg t ≤ 1))
    case True
    then show ?thesis by simp
  next
    case False
    then show ?thesis using Node.IH R.ranked-dtree-rec by auto
  qed
qed

lemma merge1-dlverts-eq[simp]: dlverts (merge1 t) = dlverts t
using ranked-dtree-axioms proof(induction t)
  case (Node r xs)
  then interpret R: ranked-dtree Node r xs rank by blast
  show ?case
  proof(cases fcard xs > 1 ∧ (∀ t ∈ fst ‘ fset xs. max-deg t ≤ 1))
    case True
    then show ?thesis by simp
  next
    case False
    then show ?thesis using Node.IH R.ranked-dtree-rec by auto
  qed
qed

lemma dverts-merge1-img-sub:
  ∀ (t2,e2) ∈ fset xs. dverts (merge1 t2) ⊆ dverts t2
    ⟹ dverts (Node r ((λ(t,e). (merge1 t,e)) |‘| xs)) ⊆ dverts (Node r xs)
  by fastforce

lemma merge1-dverts-sub: dverts (merge1 t1) ⊆ dverts t1
proof(induction t1)
  case (Node r xs)
  show ?case
  proof(cases fcard xs > 1 ∧ (∀ t ∈ fst ‘ fset xs. max-deg t ≤ 1))
    case True
    then show ?thesis using dverts-merge-sub by force
  next
    case False
    then have ∀ (t2,e2) ∈ fset xs. dverts (merge1 t2) ⊆ dverts t2 using Node by
fastforce
    then show ?thesis using False dverts-merge1-img-sub by auto
  qed
qed

lemma disjoint-dlverts-merge1: disjoint-dlverts ((λ(t,e). (merge1 t,e)) |‘| (sucs t))
proof −
  have ∀ (t, e)∈fset (sucs t). dlverts (merge1 t) ⊆ dlverts t
    using ranked-dtree.merge1-dlverts-eq ranked-dtree-rec[of root t] by force
```

**then show** *?thesis* **using** *disjoint-dlverts-img*[*OF disjoint-dlverts-if-wf*[*OF wf-lverts*]]
**by** *simp*
**qed**

**lemma** *root-empty-inter-dlverts-merge1*:
**assumes** (*x1*,*e1*) ∈ *fset* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| (*sucs t*))
**shows** *set* (*root t*) ∩ *dlverts x1* = {}
**proof** (*rule ccontr*)
**assume** *asm*: *set* (*root t*) ∩ *dlverts x1* ≠ {}
**obtain** *x2* **where** *x2-def*: *merge1 x2* = *x1* (*x2*,*e1*) ∈ *fset* (*sucs t*) **using** *assms*
**by** *auto*
**then interpret** *X*: *ranked-dtree x2* **using** *ranked-dtree-rec dtree.collapse* **by** *blast*
**have** *set* (*root t*) ∩ *dlverts x2* ≠ {} **using** *X.merge1-dlverts-eq x2-def*(*1*) *asm*
**by** *argo*
**then show** *False* **using** *x2-def*(*2*) *wf-lverts wf-dlverts.simps*[*of root t sucs t*] **by**
*auto*
**qed**

**lemma** *wf-dlverts-merge1*: *wf-dlverts* (*merge1 t*)
**using** *ranked-dtree-axioms* **proof**(*induction t*)
**case** (*Node r xs*)
**then interpret** *R*: *ranked-dtree Node r xs rank* **by** *blast*
**show** *?case*
**proof**(*cases fcard xs* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*))
**case** *True*
**then show** *?thesis* **using** *R.merge-wf-dlverts* **by** *simp*
**next**
**case** *False*
**have** (∀ (*t*,*e*) ∈ *fset* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs*). *set r* ∩ *dlverts t* = {} ∧
*wf-dlverts t*)
**using** *R.ranked-dtree-rec Node.IH R.root-empty-inter-dlverts-merge1* **by** *fast-*
*force*
**then show** *?thesis* **using** *R.disjoint-dlverts-merge1 R.wf-lverts False* **by** *auto*
**qed**
**qed**

**lemma** *merge1-darcs-eq*[*simp*]: *darcs* (*merge1 t*) = *darcs t*
**using** *ranked-dtree-axioms* **proof**(*induction t*)
**case** (*Node r xs*)
**then interpret** *R*: *ranked-dtree Node r xs rank* **by** *blast*
**show** *?case* **using** *Node.IH R.ranked-dtree-rec* **by** *auto*
**qed**

**lemma** *disjoint-darcs-merge1*: *disjoint-darcs* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| (*sucs t*))
**proof** −
**have** ∀ (*t*, *e*)∈*fset* (*sucs t*). *darcs* (*merge1 t*) ⊆ *darcs t*
**using** *ranked-dtree.merge1-darcs-eq ranked-dtree-rec*[*of root t*] **by** *force*
**then show** *?thesis* **using** *disjoint-darcs-img*[*OF disjoint-darcs-if-wf*[*OF wf-arcs*]]
**by** *simp*

**qed**

**lemma** *wf-darcs-merge1*: *wf-darcs* (*merge1 t*)
**using** *ranked-dtree-axioms* **proof**(*induction t*)
**case** (*Node r xs*)
**then interpret** *R*: *ranked-dtree Node r xs rank* **by** *blast*
**show** *?case*
**proof**(*cases fcard xs* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs. max-deg t* ≤ *1*))
**case** *True*
**then show** *?thesis* **using** *R.merge-wf-darcs* **by** *simp*
**next**
**case** *False*
**then show** *?thesis*
**using** *R.disjoint-darcs-merge1 R.ranked-dtree-rec Node.IH*
**by** (*auto simp*: *wf-darcs-iff-darcs'*)
**qed**
**qed**

**theorem** *ranked-dtree-merge1*: *ranked-dtree* (*merge1 t*) *cmp*
**by**(*unfold-locales*) (*auto simp*: *wf-darcs-merge1 wf-dlverts-merge1 dest*: *cmp-antisym*)

**lemma** *distinct-merge1*:
⟦∀ *v*∈*dverts t. distinct v*; *v*∈*dverts* (*merge1 t*)⟧ ⟹ *distinct v*
**using** *ranked-dtree-axioms* **proof**(*induction t arbitrary*: *v rule*: *merge1.induct*)
**case** (*1 r xs*)
**then interpret** *R*: *ranked-dtree Node r xs rank* **by** *blast*
**show** *?case*
**proof**(*cases fcard xs* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs. max-deg t* ≤ *1*))
**case** *True*
**then show** *?thesis* **using** *R.distinct-merge*[*OF 1.prems*(*1*)] *1.prems*(*2*) **by** *simp*
**next**
**case** *ind*: *False*
**then show** *?thesis*
**proof**(*cases v* = *r*)
**case** *False*
**have** *v*∈*dverts* (*merge1* (*Node r xs*)) ⟷ *v* ∈ *dverts* (*Node r* ((λ(*t*,*e*). (*merge1*
*t*,*e*)) |‘| *xs*))
**using** *ind* **by** *auto*
**then obtain** *t e* **where** *t-def*: (*t*,*e*) ∈ *fset xs v* ∈ *dverts* (*merge1 t*)
**using** *False 1.prems*(*2*) **by** *auto*
**then have** ∀ *v*∈*dverts t. distinct v* **using** *1.prems*(*1*) **by** *force*
**then show** *?thesis* **using** *1.IH*[*OF ind*] *t-def R.ranked-dtree-rec* **by** *fast*
**qed**(*simp add*: *1.prems*(*1*))
**qed**
**qed**

**lemma** *merge1-root-eq*[*simp*]: *root* (*merge1 t1*) = *root t1*
**by**(*induction t1*) *simp*

**lemma** *merge1-hd-root-eq*[*simp*]: *hd* (*root* (*merge1 t1*)) = *hd* (*root t1*)
**by** *simp*

**lemma** *merge1-mdeg-le*: *max-deg* (*merge1 t1*) ≤ *max-deg t1*
**proof**(*induction t1*)
**case** (*Node r xs*)
**then show** *?case*
**proof**(*cases fcard xs* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*))
**case** *True*
**then have** *max-deg* (*merge1* (*Node r xs*)) ≤ *1* **using** *merge-mdeg-le-1* **by** *simp*
**then show** *?thesis* **using** *mdeg-ge-fcard*[*of xs*] *True* **by** *simp*
**next**
**case** *False*
**have** *0*: ∀ (*t*,*e*) ∈ *fset xs*. *max-deg* (*merge1 t*) ≤ *max-deg t* **using** *Node* **by** *force*
**have** *merge1* (*Node r xs*) = (*Node r* ((λ(*t*, *e*). (*merge1 t*, *e*)) |‘| *xs*))
**using** *False* **by** *auto*
**then show** *?thesis* **using** *mdeg-img-le'*[*OF 0*] **by** *simp*
**qed**
**qed**

**lemma** *merge1-childdeg-gt1-if-fcard-gt1*:
*fcard* (*sucs* (*merge1 t1*)) > *1* ⟹ ∃ *t* ∈ *fst* ‘ *fset* (*sucs t1*). *max-deg t* > *1*
**proof**(*induction t1*)
**case** (*Node r xs*)
**have** *0*: ¬(*fcard xs* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*))
**using** *merge-fcard-le1*[*of Node r xs*] *Node.prems*(*1*) **by** *fastforce*
**then have** *fcard* (*sucs* (*merge1* (*Node r xs*))) ≤ *fcard xs* **using** *fcard-image-le*
**by** *auto*
**then show** *?case* **using** *0 Node.prems*(*1*) **by** *fastforce*
**qed**

**lemma** *merge1-fcard-le*: *fcard* (*sucs* (*merge1* (*Node r xs*))) ≤ *fcard xs*
**using** *fcard-image-le merge-fcard-le1*[*of Node r xs*] **by** *auto*

**lemma** *merge1-subtree-if-fcard-gt1*:
⟦*is-subtree* (*Node r xs*) (*merge1 t1*); *fcard xs* > *1*⟧
⟹ ∃ *ys*. *merge1* (*Node r ys*) = *Node r xs* ∧ *is-subtree* (*Node r ys*) *t1* ∧ *fcard*
*xs* ≤ *fcard ys*
**proof**(*induction t1*)
**case** (*Node r1 xs1*)
**have** *0*: ¬(*fcard xs1* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs1*. *max-deg t* ≤ *1*))
**using** *merge-fcard-le1-sub Node.prems* **by** *fastforce*
**then have** *eq*: *merge1* (*Node r1 xs1*) = *Node r1* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs1*)
**by** *auto*
**show** *?case*
**proof**(*cases Node r xs* = *merge1* (*Node r1 xs1*))
**case** *True*
**moreover have** *r* = *r1* **using** *True eq* **by** *auto*
**moreover have** *fcard xs* ≤ *fcard xs1* **using** *merge1-fcard-le True dtree.sel*(*2*)[*of*

*r xs*] **by** *auto*
  **ultimately show** *?thesis* **using** *self-subtree Node.prems(2)* **by** *auto*
 **next**
  **case** *False*
  **then obtain** *t2 e2* **where** (*t2*,*e2*) ∈ *fset xs1 is-subtree* (*Node r xs*) (*merge1 t2*)
   **using** *eq Node.prems(1)* **by** *auto*
  **then show** *?thesis* **using** *Node.IH*[*of* (*t2*,*e2*) *t2*] *Node.prems(2)* **by** *fastforce*
 **qed**
**qed**

**lemma** *merge1-childdeg-gt1-if-fcard-gt1-sub*:
 ⟦*is-subtree* (*Node r xs*) (*merge1 t1*); *fcard xs* > *1*⟧
  ⟹ ∃ *ys. merge1* (*Node r ys*) = *Node r xs* ∧ *is-subtree* (*Node r ys*) *t1*
      ∧ (∃ *t* ∈ *fst* ‘ *fset ys. max-deg t* > *1*)
 **using** *merge1-subtree-if-fcard-gt1 merge1-childdeg-gt1-if-fcard-gt1 dtree.sel(2)* **by** *metis*

**lemma** *merge1-img-eq*: ∀ (*t2*,*e2*) ∈ *fset xs. merge1 t2* = *t2* ⟹ ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs*) = *xs*
 **using** *fset-img-eq*[*of xs* λ(*t*,*e*). (*merge1 t*,*e*)] **by** *force*

**lemma** *merge1-wedge-if-uneq*:
 *merge1 t1* ≠ *t1*
  ⟹ ∃ *r xs. is-subtree* (*Node r xs*) *t1* ∧ *fcard xs* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs. max-deg t* ≤ *1*)
**proof**(*induction t1*)
 **case** (*Node r xs*)
 **show** *?case*
 **proof**(*cases fcard xs* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs. max-deg t* ≤ *1*))
  **case** *True*
  **then show** *?thesis* **by** *auto*
 **next**
  **case** *False*
  **then have** *merge1* (*Node r xs*) = *Node r* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs*) **by** *auto*
  **then obtain** *t2 e2* **where** (*t2*,*e2*) ∈ *fset xs merge1 t2* ≠ *t2*
   **using** *Node.prems merge1-img-eq*[*of xs*] **by** *auto*
  **then show** *?thesis* **using** *Node.IH*[*of* (*t2*,*e2*)] **by** *auto*
 **qed**
**qed**

**lemma** *merge1-mdeg-gt1-if-uneq*:
 **assumes** *merge1 t1* ≠ *t1*
 **shows** *max-deg t1* > *1*
**proof** −
 **obtain** *r xs* **where** *r-def*: *is-subtree* (*Node r xs*) *t1 1* < *fcard xs*
  **using** *merge1-wedge-if-uneq*[*OF assms*] **by** *fast*
 **then show** *?thesis* **using** *mdeg-ge-fcard*[*of xs*] *mdeg-ge-sub* **by** *force*
**qed**

**corollary** *merge1-eq-if-mdeg-le1*: *max-deg t1* $\leq$ *1* $\Longrightarrow$ *merge1 t1* = *t1*
**using** *merge1-mdeg-gt1-if-uneq* **by** *fastforce*

**lemma** *merge1-not-merge-if-fcard-gt1*:
⟦*merge1* (*Node r ys*) = *Node r xs*; *fcard xs* > *1*⟧ $\Longrightarrow$ *merge* (*Node r ys*) $\neq$ *Node r xs*
**using** *merge-fcard-le1*[*of Node r ys*] **by** *auto*

**lemma** *merge1-img-if-not-merge*:
*merge1* (*Node r xs*) $\neq$ *merge* (*Node r xs*)
$\Longrightarrow$ *merge1* (*Node r xs*) = *Node r* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs*)
**by** *auto*

**lemma** *merge1-img-if-fcard-gt1*:
⟦*merge1* (*Node r ys*) = *Node r xs*; *fcard xs* > *1*⟧
$\Longrightarrow$ *merge1* (*Node r ys*) = *Node r* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *ys*)
**using** *merge1-img-if-not-merge merge1-not-merge-if-fcard-gt1*[*of r ys*] **by** *simp*

**lemma** *merge1-elem-in-img-if-fcard-gt1*:
⟦*merge1* (*Node r ys*) = *Node r xs*; *fcard xs* > *1*; (*t2*,*e2*) ∈ *fset xs*⟧
$\Longrightarrow$ ∃ *t1*. (*t1*,*e2*) ∈ *fset ys* ∧ *merge1 t1* = *t2*
**using** *merge1-img-if-fcard-gt1* **by** *fastforce*

**lemma** *child-mdeg-gt1-if-sub-fcard-gt1*:
⟦*is-subtree* (*Node r xs*) (*Node v ys*); *Node r xs* $\neq$ *Node v ys*; *fcard xs* > *1*⟧
$\Longrightarrow$ ∃ *t1 e2*. (*t1*,*e2*) ∈ *fset ys* ∧ *max-deg t1* > *1*
**using** *mdeg-ge-fcard*[*of xs*] *mdeg-ge-sub* **by** *force*

**lemma** *merge1-subtree-if-mdeg-gt1*:
⟦*is-subtree* (*Node r xs*) (*merge1 t1*); *max-deg* (*Node r xs*) > *1*⟧
$\Longrightarrow$ ∃ *ys*. *merge1* (*Node r ys*) = *Node r xs* ∧ *is-subtree* (*Node r ys*) *t1*
**proof**(*induction t1*)
**case** (*Node r1 xs1*)
**then have** *0*: ¬(*fcard xs1* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs1*. *max-deg t* $\leq$ *1*))
**using** *merge-mdeg-le1-sub* **by** *fastforce*
**then have** *eq*: *merge1* (*Node r1 xs1*) = *Node r1* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs1*)
**by** *auto*
**show** *?case*
**proof**(*cases Node r xs* = *merge1* (*Node r1 xs1*))
**case** *True*
**moreover have** *r* = *r1* **using** *True eq* **by** *auto*
**moreover have** *fcard xs* $\leq$ *fcard xs1* **using** *merge1-fcard-le True dtree.sel*(*2*)[*of r xs*] **by** *auto*
**ultimately show** *?thesis* **using** *self-subtree Node.prems*(*2*) **by** *auto*
**next**
**case** *False*
**then obtain** *t2 e2* **where** (*t2*,*e2*) ∈ *fset xs1 is-subtree* (*Node r xs*) (*merge1 t2*)
**using** *eq Node.prems*(*1*) **by** *auto*
**then show** *?thesis* **using** *Node.IH*[*of* (*t2*,*e2*) *t2*] *Node.prems*(*2*) **by** *fastforce*

```
  qed
qed

lemma merge1-child-in-orig:
  assumes merge1 (Node r ys) = Node r xs and (t1,e1) ∈ fset xs
  shows ∃ t2. (t2,e1) ∈ fset ys ∧ root t2 = root t1
proof(cases fcard ys > 1 ∧ (∀ t ∈ fst ‘ fset ys. max-deg t ≤ 1))
  case True
  then show ?thesis using merge-child-in-orig[of t1 e1 Node r ys] assms by auto
next
  case False
  then have merge1 (Node r ys) = Node r ((λ(t,e). (merge1 t,e)) |‘| ys) by auto
  then show ?thesis using assms by fastforce
qed

lemma dverts-if-subtree-merge1:
  is-subtree (Node r xs) (merge1 t1) ⟹ r ∈ dverts t1
  using merge1-dverts-sub dverts-subtree-subset by fastforce

lemma subtree-merge1-orig:
  is-subtree (Node r xs) (merge1 t1) ⟹ ∃ ys. is-subtree (Node r ys) t1
  using dverts-if-subtree-merge1 subtree-root-if-dverts by fast

lemma merge1-subtree-dlverts-supset:
  is-subtree (Node r xs) (merge1 t)
    ⟹ ∃ ys. is-subtree (Node r ys) t ∧ dlverts (Node r ys) ⊆ dlverts (Node r xs)
using ranked-dtree-axioms proof(induction t)
  case (Node r1 xs1)
  then interpret R: ranked-dtree Node r1 xs1 by simp
  show ?case
  proof(cases Node r xs = merge1 (Node r1 xs1))
    case True
   then have dlverts (Node r1 xs1) ⊆ dlverts (Node r xs) using R.merge1-dlverts-eq
by simp
    moreover have r = r1 using True dtree.sel(1)[of r xs] by auto
    ultimately show ?thesis by auto
  next
    case uneq: False
    show ?thesis
    proof(cases fcard xs1 > 1 ∧ (∀ t ∈ fst ‘ fset xs1. max-deg t ≤ 1))
      case True
     then show ?thesis using R.merge-subtree-dlverts-supset Node.prems by simp
    next
      case False
       then have eq: merge1 (Node r1 xs1) = Node r1 ((λ(t,e). (merge1 t,e)) |‘|
xs1) by auto
      then obtain t2 e2 where (t2,e2) ∈ fset xs1 is-subtree (Node r xs) (merge1
t2)
        using Node.prems(1) uneq by auto
```

**then show** *?thesis* **using** *Node.IH*[*of* (*t2*,*e2*)] *R.ranked-dtree-rec* **by** *auto*
**qed**
**qed**
**qed**

**end**

## 9.5 IKKBZ-Sub

**function** *denormalize* :: ('*a list*, '*b*) *dtree* ⇒ '*a list* **where**
*denormalize* (*Node r* {|(*t*,*e*)|}) = *r* @ *denormalize t*
| ∀ *x*. *xs* ≠ {|*x*|} ⟹ *denormalize* (*Node r xs*) = *r*
**using** *dtree-to-list.cases* **by** *blast+*
**termination by** *lexicographic-order*

**lemma** *denormalize-set-eq-dlverts*: *max-deg t1* ≤ *1* ⟹ *set* (*denormalize t1*) = *dlverts t1*
**proof**(*induction t1 rule*: *denormalize.induct*)
**case** (*1 r t e*)
**then show** *?case* **using** *mdeg-ge-child*[*of t e* {|(*t*, *e*)|}] **by** *force*
**next**
**case** (*2 xs r*)
**then have** *max-deg* (*Node r xs*) = *0* **using** *mdeg-1-singleton*[*of r xs*] **by** *fastforce*
**then have** *xs* = {||} **by** (*auto intro*!: *empty-if-mdeg-0*)
**then show** *?case* **using** *2* **by** *auto*
**qed**

**lemma** *denormalize-set-sub-dlverts*: *set* (*denormalize t1*) ⊆ *dlverts t1*
**by**(*induction t1 rule*: *denormalize.induct*) *auto*

**lemma** *denormalize-distinct*:
⟦∀ *v* ∈ *dverts t1*. *distinct v*; *wf-dlverts t1*⟧ ⟹ *distinct* (*denormalize t1*)
**proof**(*induction t1 rule*: *denormalize.induct*)
**case** (*1 r t e*)
**then have** *set r* ∩ *set* (*denormalize t*) = {} **using** *denormalize-set-sub-dlverts*
**by** *fastforce*
**then show** *?case* **using** *1* **by** *auto*
**next**
**case** (*2 xs r*)
**then show** *?case* **by** *simp*
**qed**

**lemma** *denormalize-hd-root*:
**assumes** *root t* ≠ []
**shows** *hd* (*denormalize t*) = *hd* (*root t*)
**proof**(*cases* ∀ *x*. *sucs t* ≠ {|*x*|})
**case** *True*
**then show** *?thesis* **using** *denormalize.simps*(*2*)[*of sucs t root t*] **by** *simp*
**next**

**case** *False*
**then obtain** *t1 e* **where** $\{|(t1,\ e)|\} = sucs\ t$ **by** *auto*
**then show** *?thesis* **using** *denormalize.simps(1)[of root t t1 e] assms* **by** *simp*
**qed**

**lemma** *denormalize-hd-root-wf*: *wf-dlverts t* $\Longrightarrow$ *hd* (*denormalize t*) = *hd* (*root t*)
**using** *denormalize-hd-root empty-notin-wf-dlverts dtree.set-sel(1)[of t]* **by** *force*

**lemma** *denormalize-nempty-if-wf*: *wf-dlverts t* $\Longrightarrow$ *denormalize t* $\neq$ []
**by** (*induction t rule*: *denormalize.induct*) *auto*

**context** *ranked-dtree*
**begin**

**lemma** *fcard-normalize-img-if-disjoint*:
*disjoint-darcs xs* $\Longrightarrow$ *fcard* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |`| *xs*) = *fcard xs*
**using** *snds-neq-img-card-eq[of xs]* **by** *fast*

**lemma** *fcard-merge1-img-if-disjoint*:
*disjoint-darcs xs* $\Longrightarrow$ *fcard* ((λ(*t*,*e*). (*merge1 t*,*e*)) |`| *xs*) = *fcard xs*
**using** *snds-neq-img-card-eq[of xs]* **by** *fast*

**lemma** *fsts-uneq-if-disjoint-lverts-nempty*:
⟦*disjoint-dlverts xs*; ∀(*t*, *e*)∈*fset xs*. *dlverts t* $\neq$ {}⟧
$\Longrightarrow$ ∀(*t*, *e*)∈*fset xs*. ∀(*t2*, *e2*)∈*fset xs*. *t* $\neq$ *t2* ∨ (*t*, *e*) = (*t2*, *e2*)
**by** *fast*

**lemma** *normalize1-dlverts-nempty*:
∀(*t*, *e*)∈*fset xs*. *dlverts t* $\neq$ {}
$\Longrightarrow$ ∀(*t*, *e*)∈*fset* ((λ(*t*, *e*). (*normalize1 t*, *e*)) |`| *xs*). *dlverts t* $\neq$ {}
**by** *auto*

**lemma** *normalize1-fsts-uneq*:
**assumes** *disjoint-dlverts xs* **and** ∀(*t*, *e*)∈*fset xs*. *dlverts t* $\neq$ {}
**shows** ∀(*t*, *e*)∈*fset xs*. ∀(*t2*, *e2*)∈*fset xs*. *normalize1 t* $\neq$ *normalize1 t2* ∨ (*t*,*e*)
= (*t2*,*e2*)
**by** (*smt* (*verit*) *assms Int-absorb case-prodD case-prodI2 normalize1-dlverts-eq*)

**lemma** *fcard-normalize-img-if-disjoint-lverts*:
⟦*disjoint-dlverts xs*; ∀(*t*, *e*)∈*fset xs*. *dlverts t* $\neq$ {}⟧
$\Longrightarrow$ *fcard* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |`| *xs*) = *fcard xs*
**using** *fst-neq-img-card-eq[of xs normalize1] normalize1-fsts-uneq* **by** *auto*

**lemma** *fcard-normalize-img-if-wf-dlverts*:
*wf-dlverts* (*Node r xs*) $\Longrightarrow$ *fcard* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |`| *xs*) = *fcard xs*
**using** *dlverts-nempty-if-wf fcard-normalize-img-if-disjoint-lverts[of xs]* **by** *force*

**lemma** *fcard-normalize-img-if-wf-dlverts-sucs*:
*wf-dlverts t1* $\Longrightarrow$ *fcard* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |`| (*sucs t1*)) = *fcard* (*sucs t1*)

**using** *fcard-normalize-img-if-wf-dlverts*[*of root t1 sucs t1*] **by** *simp*

**lemma** *singleton-normalize1*:
**assumes** *disjoint-darcs xs* **and** $\forall x.\ xs \neq \{|x|\}$
**shows** $\forall x.\ (\lambda(t,e).\ (normalize1\ t,e))\ |`|\ xs \neq \{|x|\}$
**proof** (*rule ccontr*)
**assume** $\neg(\forall x.\ (\lambda(t,e).\ (normalize1\ t,e))\ |`|\ xs \neq \{|x|\})$
**then obtain** *x* **where** $(\lambda(t,e).\ (normalize1\ t,e))\ |`|\ xs = \{|x|\}$ **by** *blast*
**then have** *fcard* $((\lambda(t,e).\ (normalize1\ t,e))\ |`|\ xs) = 1$ **using** *fcard-single-1* **by** *force*
**then have** *fcard xs* $= 1$ **using** *fcard-normalize-img-if-disjoint*[*OF assms*(*1*)] **by** *simp*
**then have** $\exists x.\ xs = \{|x|\}$ **using** *fcard-single-1-iff* **by** *fast*
**then show** *False* **using** *assms*(*2*) **by** *simp*
**qed**

**lemma** *num-leaves-normalize1-eq*[*simp*]: *wf-darcs t1* $\Longrightarrow$ *num-leaves* (*normalize1 t1*) = *num-leaves t1*
**proof**(*induction t1*)
**case** (*Node r xs*)
**then show** *?case*
**proof**(*cases* $\forall x.\ xs \neq \{|x|\}$)
**case** *True*
**have** *fcard* $((\lambda(t,e).\ (normalize1\ t,e))\ |`|\ xs) = fcard\ xs$
**using** *fcard-normalize-img-if-disjoint Node.prems*
**by** (*auto simp*: *wf-darcs-iff-darcs'*)
**moreover have** $\forall t \in fst\ `\ fset\ xs.\ num\text{-}leaves\ (normalize1\ t) = num\text{-}leaves\ t$
**using** *Node* **by** *fastforce*
**ultimately show** *?thesis* **using** *Node sum-img-eq*[*of xs*] *True* **by** *force*
**next**
**case** *False*
**then obtain** *t e* **where** *t-def*: $xs = \{|(t,e)|\}$ **by** *auto*
**show** *?thesis*
**proof**(*cases rank* (*rev* (*root t*)) < *rank* (*rev r*))
**case** *True*
**then show** *?thesis*
**using** *t-def num-leaves-singleton num-leaves-root*[*of root t sucs t*] **by** *simp*
**next**
**case** *False*
**then show** *?thesis*
**using** *num-leaves-singleton t-def Node* **by** (*simp add*: *wf-darcs-iff-darcs'*)
**qed**
**qed**
**qed**

**lemma** *num-leaves-normalize-eq*[*simp*]: *wf-darcs t1* $\Longrightarrow$ *num-leaves* (*normalize t1*) = *num-leaves t1*
**proof**(*induction t1 rule*: *normalize.induct*)
**case** (*1 t*)

**then have** *num-leaves* (*normalize1 t*) = *num-leaves t* **using** *num-leaves-normalize1-eq*
**by** *blast*
**then show** *?case* **using** *1 wf-darcs-normalize1* **by** (*smt* (*verit*, *best*) *normalize.simps*)
**qed**

**lemma** *num-leaves-normalize1-le*: *num-leaves* (*normalize1 t1*) ≤ *num-leaves t1*
**proof**(*induction t1*)
**case** (*Node r xs*)
**then show** *?case*
**proof**(*cases* ∀ *x*. *xs* ≠ {|*x*|})
**case** *True*
**have** *fcard-le*: *fcard* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs*) ≤ *fcard xs*
**by** (*simp add*: *fcard-image-le*)
**moreover have** *xs-le*: ∀ *t*∈*fst* ‘ *fset xs*. *num-leaves* (*normalize1 t*) ≤ *num-leaves t*
**using** *Node* **by** *fastforce*
**ultimately show** *?thesis* **using** *Node sum-img-le*[*of xs*] *xs-le* ‹∀ *x*. *xs* ≠ {|*x*|}›
**by** *simp*
**next**
**case** *False*
**then obtain** *t e* **where** *t-def*: *xs* = {|(*t*,*e*)|} **by** *auto*
**show** *?thesis*
**proof**(*cases rank* (*rev* (*root t*)) < *rank* (*rev r*))
**case** *True*
**then show** *?thesis*
**using** *t-def num-leaves-singleton num-leaves-root*[*of root t sucs t*] **by** *simp*
**next**
**case** *False*
**then show** *?thesis* **using** *num-leaves-singleton t-def Node* **by** *simp*
**qed**
**qed**
**qed**

**lemma** *num-leaves-normalize-le*: *num-leaves* (*normalize t1*) ≤ *num-leaves t1*
**proof**(*induction t1 rule*: *normalize.induct*)
**case** (*1 t*)
**then have** *num-leaves* (*normalize1 t*) ≤ *num-leaves t* **using** *num-leaves-normalize1-le*
**by** *blast*
**then show** *?case* **using** *1* **by** (*smt* (*verit*) *le-trans normalize.simps*)
**qed**

**lemma** *num-leaves-merge1-le*: *num-leaves* (*merge1 t1*) ≤ *num-leaves t1*
**proof**(*induction t1*)
**case** (*Node r xs*)
**then show** *?case*
**proof**(*cases fcard xs* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*))
**case** *True*
**then have** *merge1* (*Node r xs*) = *merge* (*Node r xs*) **by** *simp*

```
    then have num-leaves (merge1 (Node r xs)) = 1
      unfolding merge-def using dtree-from-list-1-leaf by fastforce
    also have ... < fcard xs using True by blast
    also have ... ≤ num-leaves (Node r xs) using num-leaves-ge-card by fast
    finally show ?thesis by simp
  next
    case False
    have ∀ t ∈ fst ‘ fset xs. num-leaves (merge1 t) ≤ num-leaves t using Node by
force
    then show ?thesis using sum-img-le False by auto
  qed
qed

lemma num-leaves-merge1-lt: max-deg t1 > 1 ⟹ num-leaves (merge1 t1) <
num-leaves t1
proof(induction t1)
  case (Node r xs)
  show ?case
  proof(cases fcard xs > 1 ∧ (∀ t ∈ fst ‘ fset xs. max-deg t ≤ 1))
    case True
    then have merge1 (Node r xs) = merge (Node r xs) by simp
    then have num-leaves (merge1 (Node r xs)) = 1
      unfolding merge-def using dtree-from-list-1-leaf by fastforce
    also have ... < fcard xs using True by blast
    finally show ?thesis using num-leaves-ge-card less-le-trans by fast
  next
    case False
   have 0: xs ≠ {||} using Node.prems by (metis nempty-if-mdeg-n0 not-one-less-zero)
    have 1: ∀ t ∈ fst ‘ fset xs. num-leaves (merge1 t) ≤ num-leaves t
      using num-leaves-merge1-le by blast
   have ∃ t ∈ fst ‘ fset xs. max-deg t > 1 using Node.prems False mdeg-child-if-wedge
by auto
    then have 2: ∃ t ∈ fst ‘ fset xs. num-leaves (merge1 t) < num-leaves t using
Node.IH by force
    have 3: ∀ t∈fst ‘ fset xs. 0 < num-leaves t
      using num-leaves-ge1 by (metis neq0-conv not-one-le-zero)
    from False have merge1 (Node r xs) = Node r ((λ(t,e). (merge1 t,e)) |‘| xs)
by auto
    then have num-leaves (merge1 (Node r xs))
          = (∑ (t,e)∈ fset ((λ(t,e). (merge1 t,e)) |‘| xs). num-leaves t) using 0 by
auto
    then show ?thesis using 0 sum-img-lt[OF 1 2 3] by simp
  qed
qed

lemma ikkbz-num-leaves-decr:
  max-deg t1 > 1 ⟹ num-leaves (merge1 (normalize t1)) < num-leaves t1
 using num-leaves-merge1-lt num-leaves-normalize-le num-leaves-1-if-mdeg-1 num-leaves-ge1
 by (metis antisym-conv2 dual-order.antisym dual-order.trans not-le-imp-less num-leaves-merge1-le)
```

**function** *ikkbz-sub* :: (*'a list*,*'b*) *dtree* ⇒ (*'a list*,*'b*) *dtree* **where**
*ikkbz-sub t1* = (*if max-deg t1* ≤ *1 then t1 else ikkbz-sub* (*merge1* (*normalize t1*)))
**by** *auto*
**termination using** *ikkbz-num-leaves-decr* **by**(*relation measure* (λ*t*. *num-leaves t*))
*auto*

**lemma** *ikkbz-sub-darcs-sub*: *darcs* (*ikkbz-sub t*) ⊆ *darcs t*
**using** *ranked-dtree-axioms* **proof**(*induction t rule*: *ikkbz-sub.induct*)
**case** (*1 t*)
**show** *?case*
**proof**(*cases max-deg t* ≤ *1*)
**case** *False*
**have** *darcs* (*merge1* (*normalize t*)) = *darcs* (*normalize t*)
**using** *ranked-dtree.merge1-darcs-eq ranked-dtree.ranked-dtree-normalize 1.prems*
**by** *blast*
**moreover have** *ranked-dtree* (*merge1* (*normalize t*)) *cmp*
**using** *ranked-dtree.ranked-dtree-normalize 1.prems ranked-dtree.ranked-dtree-merge1*
**by** *blast*
**moreover have** ¬ (*max-deg t* ≤ *1* ∨ ¬ *list-dtree t*) **using** *False ranked-dtree-def*
*1.prems* **by** *blast*
**ultimately show** *?thesis* **using** *1.IH normalize-darcs-sub* **by** *force*
**qed**(*simp*)
**qed**

**lemma** *ikkbz-sub-dlverts-eq*[*simp*]: *dlverts* (*ikkbz-sub t*) = *dlverts t*
**using** *ranked-dtree-axioms* **proof**(*induction t rule*: *ikkbz-sub.induct*)
**case** (*1 t*)
**show** *?case*
**proof**(*cases max-deg t* ≤ *1*)
**case** *True*
**then show** *?thesis* **by** *simp*
**next**
**case** *False*
**then show** *?thesis*
**using** *1 ranked-dtree.merge1-dlverts-eq*[*of normalize t*] *normalize-dlverts-eq*
*ranked-dtree.ranked-dtree-normalize ranked-dtree.ranked-dtree-merge1 ikkbz-sub.elims*
**by** *metis*
**qed**
**qed**

**lemma** *ikkbz-sub-wf-darcs*: *wf-darcs* (*ikkbz-sub t*)
**using** *ranked-dtree-axioms* **proof**(*induction t rule*: *ikkbz-sub.induct*)
**case** (*1 t*)
**then show** *?case*
**proof**(*cases max-deg t* ≤ *1*)
**case** *True*
**then show** *?thesis* **using** *1.prems list-dtree-def ranked-dtree-def* **by** *auto*
**next**

```
    case False
    then show ?thesis
      using 1 ranked-dtree.ranked-dtree-normalize ranked-dtree.ranked-dtree-merge1
      by (metis ikkbz-sub.simps)
  qed
qed

lemma ikkbz-sub-wf-dlverts: wf-dlverts (ikkbz-sub t)
using ranked-dtree-axioms proof(induction t rule: ikkbz-sub.induct)
  case (1 t)
  then show ?case
  proof(cases max-deg t ≤ 1)
    case True
    then show ?thesis using 1.prems list-dtree-def ranked-dtree-def by auto
  next
    case False
    then show ?thesis
      using 1 ranked-dtree.ranked-dtree-normalize ranked-dtree.ranked-dtree-merge1
      by (metis ikkbz-sub.simps)
  qed
qed

theorem ikkbz-sub-list-dtree: list-dtree (ikkbz-sub t)
  using ikkbz-sub-wf-darcs ikkbz-sub-wf-dlverts list-dtree-def by blast

corollary ikkbz-sub-ranked-dtree: ranked-dtree (ikkbz-sub t) cmp
  using ikkbz-sub-list-dtree ranked-dtree-def ranked-dtree-axioms by blast

lemma ikkbz-sub-mdeg-le1: max-deg (ikkbz-sub t1) ≤ 1
  by (induction t1 rule: ikkbz-sub.induct) simp

corollary denormalize-ikkbz-eq-dlverts: set (denormalize (ikkbz-sub t)) = dlverts t
  using denormalize-set-eq-dlverts ikkbz-sub-mdeg-le1 ikkbz-sub-dlverts-eq by blast

lemma distinct-ikkbz-sub: ⟦∀ v∈dverts t. distinct v; v∈dverts (ikkbz-sub t)⟧ ⟹
distinct v
using list-dtree-axioms proof(induction t arbitrary: v rule: ikkbz-sub.induct)
  case (1 t)
  then interpret T1: ranked-dtree t rank cmp
    using ranked-dtree-axioms by (simp add: ranked-dtree-def)
  show ?case
   using 1 T1.ranked-dtree-normalize T1.distinct-normalize ranked-dtree.merge1-dverts-eq
     ranked-dtree.wf-dlverts-merge1 ranked-dtree.wf-darcs-merge1
    by (metis ikkbz-sub.elims list-dtree-def)
qed

corollary distinct-denormalize-ikkbz-sub:
  ∀ v∈dverts t. distinct v ⟹ distinct (denormalize (ikkbz-sub t))
  using distinct-ikkbz-sub ikkbz-sub-wf-dlverts denormalize-distinct by blast
```

**lemma** *ikkbz-sub-hd-root*[*simp*]: *hd* (*root* (*ikkbz-sub t*)) = *hd* (*root t*)
**using** *list-dtree-axioms* **proof**(*induction t rule*: *ikkbz-sub.induct*)
**case** (*1 t*)
**then interpret** *T1*: *ranked-dtree t rank cmp*
**using** *ranked-dtree-axioms* **by** (*simp add*: *ranked-dtree-def*)
**show** *?case*
**using** *1 merge1-hd-root-eq ranked-dtree.axioms*(*1*) *ranked-dtree.ranked-dtree-merge1*
**by** (*metis T1.ranked-dtree-normalize T1.wf-lverts ikkbz-sub.simps normalize-hd-root-eq'*)
**qed**

**corollary** *denormalize-ikkbz-sub-hd-root*[*simp*]: *hd* (*denormalize* (*ikkbz-sub t*)) = *hd* (*root t*)
**using** *ikkbz-sub-hd-root denormalize-hd-root*
**by** (*metis dtree.set-sel*(*1*) *empty-notin-wf-dlverts ikkbz-sub-wf-dlverts*)

**end**

**locale** *precedence-graph* = *finite-directed-tree* +
**fixes** *rank* :: *'a list* ⇒ *real*
**fixes** *cost* :: *'a list* ⇒ *real*
**fixes** *cmp* :: (*'a list*×*'b*) *comparator*
**assumes** *asi-rank*: *asi rank root cost*
**and** *cmp-antisym*:
⟦*v1* ≠ []; *v2* ≠ []; *compare cmp* (*v1*,*e1*) (*v2*,*e2*) = *Equiv*⟧ ⟹ *set v1* ∩ *set v2* ≠ {} ∨ *e1*=*e2*
**begin**

**definition** *to-list-dtree* :: (*'a list*, *'b*) *dtree* **where**
*to-list-dtree* = *finite-directed-tree.to-dtree to-list-tree* [*root*]

**lemma** *to-list-dtree-single*: *v* ∈ *dverts to-list-dtree* ⟹ ∃ *x*. *v* = [*x*] ∧ *x* ∈ *verts T*
**unfolding** *to-list-dtree-def* **using** *to-list-tree-single*
**by** (*simp add*: *finite-directed-tree.dverts-eq-verts to-list-tree-finite-directed-tree*)

**lemma** *to-list-dtree-wf-dverts*: *wf-dverts to-list-dtree*
**using** *finite-directed-tree.wf-dverts-to-dtree*[*OF to-list-tree-finite-directed-tree*]
**by**(*simp add*: *to-list-dtree-def*)

**lemma** *to-list-dtree-wf-dlverts*: *wf-dlverts to-list-dtree*
**unfolding** *to-list-dtree-def*
**by** (*simp add*: *to-list-tree-fin-list-directed-tree fin-list-directed-tree.wf-dlverts-to-dtree*)

**lemma** *to-list-dtree-wf-darcs*: *wf-darcs to-list-dtree*
**using** *finite-directed-tree.wf-darcs-to-dtree*[*OF to-list-tree-finite-directed-tree*]
**by**(*simp add*: *to-list-dtree-def*)

**lemma** *to-list-dtree-list-dtree*: *list-dtree to-list-dtree*
**by**(*simp add*: *list-dtree-def to-list-dtree-wf-dlverts to-list-dtree-wf-darcs*)

**lemma** *to-list-dtree-ranked-dtree*: *ranked-dtree to-list-dtree cmp*
**by**(*auto simp*: *ranked-dtree-def to-list-dtree-list-dtree ranked-dtree-axioms-def dest*: *cmp-antisym*)

**interpretation** *t*: *ranked-dtree to-list-dtree* **by** (*rule to-list-dtree-ranked-dtree*)

**definition** *ikkbz-sub* :: *'a list* **where**
*ikkbz-sub* = *denormalize* (*t.ikkbz-sub to-list-dtree*)

**lemma** *dverts-eq-verts-to-list-tree*: *dverts to-list-dtree* = *pre-digraph.verts to-list-tree*
**unfolding** *to-list-dtree-def*
**by** (*simp add*: *finite-directed-tree.dverts-eq-verts to-list-tree-finite-directed-tree*)

**lemma** *dverts-eq-verts-img*: *dverts to-list-dtree* = (λ*x*. [*x*]) ‘ *verts T*
**by** (*simp add*: *dverts-eq-verts-to-list-tree to-list-tree-def*)

**lemma** *dlverts-eq-verts*: *dlverts to-list-dtree* = *verts T*
**by** (*simp add*: *dverts-eq-verts-img dlverts-eq-dverts-union*)

**theorem** *ikkbz-set-eq-verts*: *set ikkbz-sub* = *verts T*
**using** *dlverts-eq-verts ikkbz-sub-def t.denormalize-ikkbz-eq-dlverts* **by** *simp*

**lemma** *distinct-to-list-tree*: ∀ *v*∈*verts to-list-tree*. *distinct v*
**unfolding** *to-list-tree-def* **by** *simp*

**lemma** *distinct-to-list-dtree*: ∀ *v*∈*dverts to-list-dtree*. *distinct v*
**using** *distinct-to-list-tree dverts-eq-verts-to-list-tree* **by** *blast*

**theorem** *distinct-ikkbz-sub*: *distinct ikkbz-sub*
**unfolding** *ikkbz-sub-def*
**using** *distinct-to-list-dtree t.distinct-denormalize-ikkbz-sub* **by** *blast*

**lemma** *to-list-dtree-root-eq-root*: *Dtree.root* (*to-list-dtree*) = [*root*]
**unfolding** *to-list-dtree-def*
**by** (*simp add*: *finite-directed-tree.to-dtree-root-eq-root to-list-tree-finite-directed-tree*)

**lemma** *to-list-dtree-hd-root-eq-root*[*simp*]: *hd* (*Dtree.root to-list-dtree*) = *root*
**by** (*simp add*: *to-list-dtree-root-eq-root*)

**theorem** *ikkbz-sub-hd-eq-root*[*simp*]: *hd ikkbz-sub* = *root*
**unfolding** *ikkbz-sub-def* **using** *t.denormalize-ikkbz-sub-hd-root to-list-dtree-root-eq-root*
**by** *simp*

**end**

## 9.6 Full IKKBZ

**locale** *tree-query-graph* = *undir-tree-todir G* + *query-graph G* **for** *G*

**locale** *cmp-tree-query-graph* = *tree-query-graph* +
  **fixes** *cmp* :: (*'a list*×*'b*) *comparator*
  **assumes** *cmp-antisym*:
    ⟦*v1* ≠ []; *v2* ≠ []; *compare cmp* (*v1*,*e1*) (*v2*,*e2*) = *Equiv*⟧ ⟹ *set v1* ∩ *set v2* ≠ {} ∨ *e1*=*e2*

**locale** *ikkbz-query-graph* = *cmp-tree-query-graph* +
  **fixes** *cost* :: *'a joinTree* ⇒ *real*
  **fixes** *cost-r* :: *'a* ⇒ (*'a list* ⇒ *real*)
  **fixes** *rank-r* :: *'a* ⇒ (*'a list* ⇒ *real*)
  **assumes** *asi-rank*: *r* ∈ *verts G* ⟹ *asi* (*rank-r r*) *r* (*cost-r r*)
    **and** *cost-correct*:
      ⟦*valid-tree t*; *no-cross-products t*; *left-deep t*⟧
        ⟹ *cost-r* (*first-node t*) (*revorder t*) = *cost t*
**begin**

**abbreviation** *ikkbz-sub* :: *'a* ⇒ *'a list* **where**
  *ikkbz-sub r* ≡ *precedence-graph.ikkbz-sub* (*dir-tree-r r*) *r* (*rank-r r*) *cmp*

**abbreviation** *cost-l* :: *'a list* ⇒ *real* **where**
  *cost-l xs* ≡ *cost* (*create-ldeep xs*)

**lemma** *precedence-graph-r*:
  *r* ∈ *verts G* ⟹ *precedence-graph* (*dir-tree-r r*) *r* (*rank-r r*) (*cost-r r*) *cmp*
  **using** *fin-directed-tree-r cmp-antisym*
  **by** (*simp add*: *precedence-graph-def precedence-graph-axioms-def asi-rank*)

**lemma** *nempty-if-set-eq-verts*: *set xs* = *verts G* ⟹ *xs* ≠ []
  **using** *verts-nempty* **by** *force*

**lemma** *revorder-if-set-eq-verts*: *set xs* = *verts G* ⟹ *revorder* (*create-ldeep xs*) = *rev xs*
  **using** *nempty-if-set-eq-verts create-ldeep-order* **unfolding** *revorder-eq-rev-inorder*
**by** *blast*

**lemma** *cost-correct'*:
  ⟦*set xs* = *verts G*; *distinct xs*; *no-cross-products* (*create-ldeep xs*)⟧
    ⟹ *cost-r* (*hd xs*) (*rev xs*) = *cost-l xs*
  **using** *cost-correct*[*of create-ldeep xs*] *revorder-if-set-eq-verts create-ldeep-ldeep*[*of xs*]
  **unfolding** *valid-tree-def distinct-relations-def*
  **by** (*simp add*: *create-ldeep-order create-ldeep-relations first-node-eq-hd nempty-if-set-eq-verts*)

**lemma** *ikkbz-sub-verts-eq*: *r* ∈ *verts G* ⟹ *set* (*ikkbz-sub r*) = *verts G*
  **using** *precedence-graph.ikkbz-set-eq-verts precedence-graph-r verts-dir-tree-r-eq* **by** *fast*

**lemma** *ikkbz-sub-distinct*: *r* ∈ *verts G* ⟹ *distinct* (*ikkbz-sub r*)

**using** *precedence-graph.distinct-ikkbz-sub precedence-graph-r* **by** *fast*

**lemma** *ikkbz-sub-hd-eq-root*: $r \in verts\ G \Longrightarrow hd\ (ikkbz\text{-}sub\ r) = r$
**using** *precedence-graph.ikkbz-sub-hd-eq-root precedence-graph-r* **by** *fast*

**definition** *ikkbz* :: *'a list* **where**
$ikkbz \equiv arg\text{-}min\text{-}on\ cost\text{-}l\ \{ikkbz\text{-}sub\ r | r.\ r \in verts\ G\}$

**lemma** *ikkbz-sub-set-fin*: *finite* $\{ikkbz\text{-}sub\ r | r.\ r \in verts\ G\}$
**by** *simp*

**lemma** *ikkbz-sub-set-nempty*: $\{ikkbz\text{-}sub\ r | r.\ r \in verts\ G\} \neq \{\}$
**by** (*simp add*: *verts-nempty*)

**lemma** *ikkbz-in-ikkbz-sub-set*: $ikkbz \in \{ikkbz\text{-}sub\ r | r.\ r \in verts\ G\}$
**unfolding** *ikkbz-def* **using** *ikkbz-sub-set-fin ikkbz-sub-set-nempty arg-min-if-finite*
**by** *blast*

**lemma** *ikkbz-eq-ikkbz-sub*: $\exists\, r \in verts\ G.\ ikkbz = ikkbz\text{-}sub\ r$
**using** *ikkbz-in-ikkbz-sub-set* **by** *blast*

**lemma** *ikkbz-min-ikkbz-sub*: $r \in verts\ G \Longrightarrow cost\text{-}l\ ikkbz \leq cost\text{-}l\ (ikkbz\text{-}sub\ r)$
**unfolding** *ikkbz-def* **using** *ikkbz-sub-set-fin arg-min-least* **by** *fast*

**lemma** *ikkbz-distinct*: *distinct ikkbz*
**using** *ikkbz-eq-ikkbz-sub ikkbz-sub-distinct* **by** *fastforce*

**lemma** *ikkbz-set-eq-verts*: *set ikkbz* = *verts G*
**using** *ikkbz-eq-ikkbz-sub ikkbz-sub-verts-eq* **by** *force*

**lemma** *ikkbz-nempty*: $ikkbz \neq []$
**using** *ikkbz-set-eq-verts verts-nempty* **by** *fastforce*

**lemma** *ikkbz-hd-in-verts*: *hd ikkbz* $\in$ *verts G*
**using** *ikkbz-nempty ikkbz-set-eq-verts* **by** *fastforce*

**lemma** *inorder-ikkbz*: *inorder* (*create-ldeep ikkbz*) = *ikkbz*
**using** *create-ldeep-order ikkbz-nempty* **by** *blast*

**lemma** *inorder-ikkbz-distinct*: *distinct* (*inorder* (*create-ldeep ikkbz*))
**using** *ikkbz-distinct inorder-ikkbz* **by** *simp*

**lemma** *inorder-relations-eq-verts*: *relations* (*create-ldeep ikkbz*) = *verts G*
**using** *ikkbz-set-eq-verts create-ldeep-relations ikkbz-nempty* **by** *blast*

**theorem** *ikkbz-valid-tree*: *valid-tree* (*create-ldeep ikkbz*)
**unfolding** *valid-tree-def distinct-relations-def*
**using** *inorder-ikkbz-distinct inorder-relations-eq-verts* **by** *blast*

**end**

**locale** *old = list-dtree t* **for** *t* :: (*'a list*,*'b*) *dtree* +
  **fixes** *rank* :: *'a list* ⇒ *real*
**begin**

**function** *find-pos-aux* :: *'a list* ⇒ *'a list* ⇒ (*'a list*,*'b*) *dtree* ⇒ (*'a list* × *'a list*)
**where**
  *find-pos-aux v p* (*Node r* {|(*t1*,-)|}) =
    (*if rank* (*rev v*) ≤ *rank* (*rev r*) *then* (*p*,*r*) *else find-pos-aux v r t1*)
| ∀ *x*. *xs* ≠ {|*x*|} ⟹ *find-pos-aux v p* (*Node r xs*) =
    (*if rank* (*rev v*) ≤ *rank* (*rev r*) *then* (*p*,*r*) *else* (*r*,*r*))
  **by** (*metis combine.cases old.prod.exhaust*) *auto*
**termination by** *lexicographic-order*

**function** *find-pos* :: *'a list* ⇒ (*'a list*,*'b*) *dtree* ⇒ (*'a list* × *'a list*) **where**
  *find-pos v* (*Node r* {|(*t1*,-)|}) = *find-pos-aux v r t1*
| ∀ *x*. *xs* ≠ {|*x*|} ⟹ *find-pos v* (*Node r xs*) = (*r*,*r*)
  **by** (*metis dtree.exhaust surj-pair*) *auto*
**termination by** *lexicographic-order*

**abbreviation** *insert-chain* :: (*'a list*×*'b*) *list* ⇒ (*'a list*,*'b*) *dtree* ⇒ (*'a list*,*'b*) *dtree*
**where**
  *insert-chain xs t1* ≡
    *foldr* (λ(*v*,*e*) *t2*. *case find-pos v t2 of* (*x*,*y*) ⇒ *insert-between v e x y t2*) *xs t1*

**fun** *merge* :: (*'a list*,*'b*) *dtree* ⇒ (*'a list*,*'b*) *dtree* **where**
  *merge* (*Node r xs*) = *ffold* (λ(*t*,*e*) *b*. *case b of Node r xs* ⇒
      *if xs* = {||} *then Node r* {|(*t*,*e*)|} *else insert-chain* (*dtree-to-list t*) *b*)
    (*Node r* {||}) *xs*

**lemma** *ffold-if-False-eq-acc*:
  ⟦∀ *a*. ¬*P a*; *comp-fun-commute* (λ*a b*. *if* ¬*P a then b else Q a b*)⟧
    ⟹ *ffold* (λ*a b*. *if* ¬*P a then b else Q a b*) *acc xs* = *acc*
**proof**(*induction xs*)
  **case** (*insert x xs*)
  **let** *?f* = λ*a b*. *if* ¬*P a then b else Q a b*
  **have** *ffold ?f acc* (*finsert x xs*) = *?f x* (*ffold ?f acc xs*)
   **using** *insert.hyps* **by** (*simp add*: *comp-fun-commute.ffold-finsert insert.prems*(*2*))
  **then have** *ffold ?f acc* (*finsert x xs*) = *ffold ?f acc xs* **using** *insert.prems* **by** *simp*
  **then show** *?case* **using** *insert.IH insert.prems* **by** *simp*
**qed**(*simp add*: *comp-fun-commute.ffold-empty*)

**lemma** *find-pos-rank-less*: *rank* (*rev v*) ≤ *rank* (*rev r*) ⟹ *find-pos-aux v p* (*Node r xs*) = (*p*,*r*)
  **by**(*cases* ∃ *x*. *xs* = {|*x*|}) *auto*

**lemma** *find-pos-y-in-dverts*: $(x,y) = \textit{find-pos-aux}\ v\ p\ t1 \Longrightarrow y \in \textit{dverts}\ t1$
**proof**(*induction t1 arbitrary*: *p*)
  **case** (*Node r xs*)
  **then show** *?case*
  **proof**(*cases rank* $(\textit{rev}\ v) \leq \textit{rank}\ (\textit{rev}\ r)$)
    **case** *True*
    **then show** *?thesis* **using** *Node.prems* **by**(*cases* $\exists\, x.\ xs = \{|x|\}$) *auto*
  **next**
    **case** *False*
    **then show** *?thesis* **using** *Node* **by**(*cases* $\exists\, x.\ xs = \{|x|\}$) *fastforce+*
  **qed**
**qed**

**lemma** *find-pos-x-in-dverts*: $(x,y) = \textit{find-pos-aux}\ v\ p\ t1 \Longrightarrow x \in \textit{dverts}\ t1 \vee p{=}x$
**proof**(*induction t1 arbitrary*: *p*)
  **case** (*Node r xs*)
  **then show** *?case*
  **proof**(*cases rank* $(\textit{rev}\ v) \leq \textit{rank}\ (\textit{rev}\ r)$)
    **case** *True*
    **then show** *?thesis* **using** *Node.prems* **by**(*cases* $\exists\, x.\ xs = \{|x|\}$) *auto*
  **next**
    **case** *False*
    **then show** *?thesis* **using** *Node* **by**(*cases* $\exists\, x.\ xs = \{|x|\}$) *fastforce+*
  **qed**
**qed**

**end**

**end**

**theory** *IKKBZ-Optimality*
 **imports** *Complex-Main CostFunctions QueryGraph IKKBZ HOL−Library.Sublist*
**begin**

# 10 Optimality of IKKBZ

**context** *directed-tree*
**begin**
**fun** *forward-arcs* :: *'a list* ⇒ *bool* **where**
  *forward-arcs* [] = *True*
| *forward-arcs* [*x*] = *True*
| *forward-arcs* (*x*#*xs*) = $((\exists\, y \in \textit{set}\ xs.\ y \rightarrow_T x) \wedge \textit{forward-arcs}\ xs)$

**fun** *no-back-arcs* :: *'a list* ⇒ *bool* **where**
  *no-back-arcs* [] = *True*
| *no-back-arcs* (*x*#*xs*) = $((\nexists\, y.\ y \in \textit{set}\ xs \wedge y \rightarrow_T x) \wedge \textit{no-back-arcs}\ xs)$

**definition** *forward* :: *'a list* ⇒ *bool* **where**
*forward xs* = (∀ *i* ∈ {*1*..(*length xs* − *1*)}. ∃ *j* < *i*. *xs*!*j* →$_T$ *xs*!*i*)

**definition** *no-back* :: *'a list* ⇒ *bool* **where**
*no-back xs* = (∄ *i j*. *i* < *j* ∧ *j* < *length xs* ∧ *xs*!*j* →$_T$ *xs*!*i*)

**definition** *seq-conform* :: *'a list* ⇒ *bool* **where**
*seq-conform xs* ≡ *forward-arcs* (*rev xs*) ∧ *no-back-arcs xs*

**definition** *before* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
*before s1 s2* ≡ *seq-conform s1* ∧ *seq-conform s2* ∧ *set s1* ∩ *set s2* = {}
∧ (∃ *x* ∈ *set s1*. ∃ *y* ∈ *set s2*. *x* →$_T$ *y*)

**definition** *before2* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
*before2 s1 s2* ≡ *seq-conform s1* ∧ *seq-conform s2* ∧ *set s1* ∩ *set s2* = {}
∧ (∃ *x* ∈ *set s1*. ∃ *y* ∈ *set s2*. *x* →$_T$ *y*)
∧ (∀ *x* ∈ *set s1*. ∀ *v* ∈ *verts T* − *set s1* − *set s2*. ¬ *x* →$_T$ *v*)

**lemma** *before-alt1*:
(∃ *i* < *length s1*. ∃ *j* < *length s2*. *s1*!*i* →$_T$ *s2*!*j*) ⟷ (∃ *x* ∈ *set s1*. ∃ *y* ∈ *set s2*. *x* →$_T$ *y*)
**using** *in-set-conv-nth* **by** *metis*

**lemma** *before-alt2*:
(∀ *i* < *length s1*. ∀ *v* ∈ *verts T* − *set s1* − *set s2*. ¬ *s1*!*i* →$_T$ *v*)
⟷ (∀ *x* ∈ *set s1*. ∀ *v* ∈ *verts T* − *set s1* − *set s2*. ¬ *x* →$_T$ *v*)
**using** *in-set-conv-nth* **by** *metis*

**lemma** *no-back-alt-aux*: (∀ *i j*. *i* ≥ *j* ∨ *j* ≥ *length xs* ∨ ¬(*xs*!*j* →$_T$ *xs*!*i*)) ⟹ *no-back xs*
**using** *less-le-not-le no-back-def* **by** *auto*

**lemma** *no-back-alt*: (∀ *i j*. *i* ≥ *j* ∨ *j* ≥ *length xs* ∨ ¬(*xs*!*j* →$_T$ *xs*!*i*)) ⟷ *no-back xs*
**using** *no-back-alt-aux* **by** (*auto simp*: *no-back-def*)

**lemma** *no-back-arcs-alt-aux1*: ⟦*no-back-arcs xs*; *i* < *j*; *j* < *length xs*⟧ ⟹ ¬(*xs*!*j* →$_T$ *xs*!*i*)
**proof**(*induction xs arbitrary*: *i j*)
**case** (*Cons x xs*)
**then show** *?case*
**proof**(*cases i* = *0*)
**case** *True*
**then show** *?thesis* **using** *Cons.prems* **by** *simp*
**next**
**case** *False*
**then show** *?thesis* **using** *Cons* **by** *auto*
**qed**
**qed**(*simp*)

**lemma** *no-back-insert-aux*:
(∀ *i j*. *i* ≥ *j* ∨ *j* ≥ *length* (*x*#*xs*) ∨ ¬((*x*#*xs*)!*j* →$_T$ (*x*#*xs*)!*i*))
⟹ (∀ *i j*. *i* ≥ *j* ∨ *j* ≥ *length xs* ∨ ¬(*xs*!*j* →$_T$ *xs*!*i*))
**by** *force*

**lemma** *no-back-insert*: *no-back* (*x*#*xs*) ⟹ *no-back xs*
**using** *no-back-alt no-back-insert-aux* **by** *blast*

**lemma** *no-arc-fst-if-no-back*:
**assumes** *no-back* (*x*#*xs*) **and** *y* ∈ *set xs*
**shows** ¬ *y* →$_T$ *x*
**proof** −
**have** *0*: (*x*#*xs*)!*0* = *x* **by** *simp*
**obtain** *j* **where** *xs*!*j* = *y j* < *length xs* **using** *assms*(*2*) **by** (*auto simp*: *in-set-conv-nth*)
**then have** (*x*#*xs*)!(*Suc j*) = *y* ∧ *Suc j* < *length* (*x*#*xs*) **by** *simp*
**then show** *?thesis* **using** *assms*(*1*) *0* **by** (*metis no-back-def zero-less-Suc*)
**qed**

**lemma** *no-back-arcs-alt-aux2*: *no-back xs* ⟹ *no-back-arcs xs*
**by**(*induction xs*) (*auto simp*: *no-back-insert no-arc-fst-if-no-back*)

**lemma** *no-back-arcs-alt*: *no-back xs* ⟷ *no-back-arcs xs*
**using** *no-back-arcs-alt-aux1 no-back-arcs-alt-aux2 no-back-alt* **by** *fastforce*

**lemma** *forward-arcs-alt-aux1*:
⟦*forward-arcs xs*; *i* ∈ {*1*..(*length* (*rev xs*) − *1*)}⟧ ⟹ ∃ *j* < *i*. (*rev xs*)!*j* →$_T$ (*rev xs*)!*i*
**proof**(*induction xs rule*: *forward-arcs.induct*)
**case** (*3 x x' xs*)
**then show** *?case*
**proof**(*cases i* = *length* (*rev* (*x*#*x'*#*xs*)) − *1*)
**case** *True*
**then have** *i*: (*rev* (*x*#*x'*#*xs*))!*i* = *x* **by** (*simp add*: *nth-append*)
**then obtain** *y* **where** *y-def*: *y*∈*set* (*x'*#*xs*) *y* →$_T$ *x* **using** *3.prems* **by** *auto*
**then obtain** *j* **where** *j-def*: *rev* (*x'*#*xs*)!*j* = *y j* < *length* (*rev* (*x'*#*xs*))
**using** *in-set-conv-nth*[*of y*] **by** *fastforce*
**then have** *rev* (*x*#*x'*#*xs*)!*j* = *y* **by** (*auto simp*: *nth-append*)
**then show** *?thesis* **using** *y-def*(*2*) *i j-def*(*2*) *True* **by** *auto*
**next**
**case** *False*
**then obtain** *j* **where** *j-def*: *j* < *i rev* (*x'* # *xs*)!*j* →$_T$ *rev* (*x'* # *xs*)!*i* **using** *3*
**by** *auto*
**then have** *rev* (*x*#*x'*#*xs*)!*j* = *rev* (*x'*#*xs*)!*j* **using** *3.prems*(*2*) **by** (*auto simp*:
*nth-append*)
**moreover have** *rev* (*x*#*x'*#*xs*)!*i* = *rev* (*x'*#*xs*)!*i*
**using** *3.prems*(*2*) *False* **by** (*auto simp*: *nth-append*)
**ultimately show** *?thesis* **using** *j-def* **by** *auto*
**qed**

**qed**(*auto*)

**lemma** *forward-split-aux*:
**assumes** *forward* (*xs*@*ys*) **and** *i*∈{*1*..*length xs* − *1*}
**shows** ∃*j*<*i*. *xs*!*j* →$_T$ *xs*!*i*
**proof** −
**obtain** *j* **where** *j* < *i* ∧ (*xs*@*ys*)!*j* →$_T$ (*xs*@*ys*)!*i* **using** *assms forward-def* **by** *force*
**moreover have** *i* < *length xs* **using** *assms*(*2*) **by** *auto*
**ultimately show** *?thesis* **by** (*auto simp*: *nth-append*)
**qed**

**lemma** *forward-split*: *forward* (*xs*@*ys*) ⟹ *forward xs*
**using** *forward-split-aux forward-def* **by** *blast*

**lemma** *forward-cons*:
*forward* (*rev* (*x*#*xs*)) ⟹ *forward* (*rev xs*)
**using** *forward-split* **by** *simp*

**lemma** *arc-to-lst-if-forward*:
**assumes** *forward* (*rev* (*x*#*xs*)) **and** *xs* = *y*#*ys*
**shows** ∃*y* ∈ *set xs*. *y* →$_T$ *x*
**proof** −
**have** (*x*#*xs*)!*0* = *x* **by** *simp*
**have** (*rev xs*@[*x*])!(*length xs*) = (*xs*@[*x*])!(*length xs*) **by** (*metis length-rev nth-append-length*)
**then have** *i*: *rev* (*x*#*xs*)!(*length xs*) = *x* **by** *simp*
**have** *length xs* ∈ {*1*..(*length* (*rev* (*x*#*xs*)) − *1*)} **using** *assms*(*2*) **by** *simp*
**then obtain** *j* **where** *j-def*: *j* < *length xs* ∧ (*rev* (*x*#*xs*))!*j* →$_T$ (*rev* (*x*#*xs*))!*length xs*
**using** *assms*(*1*) *forward-def*[*of rev* (*x*#*xs*)] **by** *blast*
**then have** *rev xs*!*j* ∈ *set xs* **using** *length-rev nth-mem set-rev* **by** *metis*
**then have** *rev* (*x*#*xs*)!*j* ∈ *set xs* **by** (*auto simp*: *j-def nth-append*)
**then show** *?thesis* **using** *i j-def* **by** *auto*
**qed**

**lemma** *forward-arcs-alt-aux2*: *forward* (*rev xs*) ⟹ *forward-arcs xs*
**proof**(*induction xs rule*: *forward-arcs.induct*)
**case** (*3 x y xs*)
**then have** *forward-arcs* (*y* # *xs*) **using** *forward-cons* **by** *blast*
**then show** *?case* **using** *arc-to-lst-if-forward 3.prems* **by** *simp*
**qed**(*auto*)

**lemma** *forward-arcs-alt*: *forward xs* ⟷ *forward-arcs* (*rev xs*)
**using** *forward-arcs-alt-aux1 forward-arcs-alt-aux2 forward-def* **by** *fastforce*

**corollary** *forward-arcs-alt′*: *forward* (*rev xs*) ⟷ *forward-arcs xs*
**using** *forward-arcs-alt* **by** *simp*

**corollary** *forward-arcs-split*: *forward-arcs* (*ys*@*xs*) ⟹ *forward-arcs xs*

**using** *forward-split*[*of rev xs rev ys*] *forward-arcs-alt* **by** *simp*

**lemma** *seq-conform-alt*: *seq-conform xs* $\longleftrightarrow$ *forward xs* $\wedge$ *no-back xs*
**using** *forward-arcs-alt no-back-arcs-alt seq-conform-def* **by** *simp*

**lemma** *forward-app-aux*:
**assumes** *forward s1 forward s2* $\exists\, x{\in}set\ s1.\ x \rightarrow_T hd\ s2\ i{\in}\{1..length\ (s1@s2) - 1\}$
**shows** $\exists\, j{<}i.\ (s1@s2)!j \rightarrow_T (s1@s2)!i$
**proof** −
**consider** $i{\in}\{1..length\ s1 - 1\} \mid i = length\ s1 \mid i{\in}\{length\ s1 + 1..length\ s1 + length\ s2 - 1\}$
**using** *assms*(*4*) **by** *fastforce*
**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then obtain** *j* **where** *j-def*: $j < i$ $s1!j \rightarrow_T s1!i$ **using** *assms*(*1*) *forward-def* **by** *blast*
**moreover have** $(s1@s2)!i = s1!i$ **using** *1* **by** (*auto simp*: *nth-append*)
**moreover have** $(s1@s2)!j = s1!j$ **using** *1 j-def*(*1*) **by** (*auto simp*: *nth-append*)
**ultimately show** *?thesis* **by** *auto*
**next**
**case** *2*
**then have** $s2 \neq []$ **using** *assms*(*4*) **by** *force*
**then have** $(s1@s2)!i = hd\ s2$ **using** *2 assms*(*4*) **by** (*simp add*: *hd-conv-nth nth-append*)
**then obtain** *x* **where** *x-def*: $x{\in}set\ s1\ x \rightarrow_T (s1@s2)!i$ **using** *assms*(*3*) **by** *force*
**then obtain** *j* **where** $s1!j = x\ j < length\ s1$ **by** (*auto simp*: *in-set-conv-nth*)
**then show** *?thesis* **using** *x-def*(*2*) *2* **by** (*auto simp*: *nth-append*)
**next**
**case** *3*
**then have** $i{-}length\ s1 \in \{1..length\ s2 - 1\}$ **by** *fastforce*
**then obtain** *j* **where** *j-def*: $j < (i{-}length\ s1)$ $s2!j \rightarrow_T s2!(i{-}length\ s1)$
**using** *assms*(*2*) *forward-def* **by** *blast*
**moreover have** $(s1@s2)!i = s2!(i{-}length\ s1)$ **using** *3* **by** (*auto simp*: *nth-append*)
**moreover have** $(s1@s2)!(j{+}length\ s1) = s2!j$ **using** *3 j-def*(*1*) **by** (*auto simp*: *nth-append*)
**ultimately have** $(j{+}length\ s1) < i \wedge (s1@s2)!(j{+}length\ s1) \rightarrow_T (s1@s2)!i$ **by** *force*
**then show** *?thesis* **by** *blast*
**qed**
**qed**

**lemma** *forward-app*: $[\![forward\ s1; forward\ s2; \exists\, x{\in}set\ s1.\ x \rightarrow_T hd\ s2]\!] \Longrightarrow forward\ (s1@s2)$
**by** (*simp add*: *forward-def forward-app-aux*)

**lemma** *before-conform1I*: *before s1 s2* $\Longrightarrow$ *seq-conform s1*

**unfolding** *before-def* **by** *blast*

**lemma** *before-forward1I*: *before s1 s2* $\Longrightarrow$ *forward s1*
**unfolding** *before-def seq-conform-alt* **by** *blast*

**lemma** *before-no-back1I*: *before s1 s2* $\Longrightarrow$ *no-back s1*
**unfolding** *before-def seq-conform-alt* **by** *blast*

**lemma** *before-ArcI*: *before s1 s2* $\Longrightarrow \exists\, x \in$ *set s1*. $\exists\, y \in$ *set s2*. $x \to_T y$
**unfolding** *before-def* **by** *blast*

**lemma** *before-conform2I*: *before s1 s2* $\Longrightarrow$ *seq-conform s2*
**unfolding** *before-def* **by** *blast*

**lemma** *before-forward2I*: *before s1 s2* $\Longrightarrow$ *forward s2*
**unfolding** *before-def seq-conform-alt* **by** *blast*

**lemma** *before-no-back2I*: *before s1 s2* $\Longrightarrow$ *no-back s2*
**unfolding** *before-def seq-conform-alt* **by** *blast*

**lemma** *hd-reach-all-forward-arcs*:
$[\![$*hd* (*rev xs*) $\in$ *verts T*; *forward-arcs xs*; $x \in$ *set xs*$]\!] \Longrightarrow$ *hd* (*rev xs*) $\to^*{}_T\ x$
**proof**(*induction xs arbitrary*: *x rule*: *forward-arcs.induct*)
**case** (*3 z y ys*)
**then have** *0*: ($\exists\, y \in$ *set* (*y*#*ys*). $y \to_T z$) *forward-arcs* (*y*#*ys*) **by** *auto*
**have** *hd-eq*: *hd* (*rev* (*z* # *y* # *ys*)) = *hd* (*rev* (*y* # *ys*))
**using** *hd-rev*[*of y*#*ys*] **by** (*auto simp*: *last-ConsR*)
**then show** *?case*
**proof**(*cases x* = *z*)
**case** *True*
**then obtain** *x'* **where** *x'-def*: $x' \in$ *set* (*y*#*ys*) $x' \to_T x$ **using** *3.prems*(*2*) **by** *auto*
**then have** *hd* (*rev* (*z* # *y* # *ys*)) $\to^*{}_T\ x'$ **using** *3 hd-eq* **by** *simp*
**then show** *?thesis* **using** *x'-def*(*2*) *reachable-adj-trans* **by** *blast*
**next**
**case** *False*
**then show** *?thesis* **using** *3 hd-eq* **by** *simp*
**qed**
**qed**(*auto*)

**lemma** *hd-reach-all-forward*:
$[\![$*hd xs* $\in$ *verts T*; *forward xs*; $x \in$ *set xs*$]\!] \Longrightarrow$ *hd xs* $\to^*{}_T\ x$
**using** *hd-reach-all-forward-arcs*[*of rev xs*] **by** (*simp add*: *forward-arcs-alt*)

**lemma** *hd-in-verts-if-forward*: *forward* (*x*#*y*#*xs*) $\Longrightarrow$ *hd* (*x*#*y*#*xs*) $\in$ *verts T*
**unfolding** *forward-def* **by** *fastforce*

**lemma** *two-elems-if-length-gt1*: *length xs* > *1* $\Longrightarrow \exists\, x\ y\ ys.\ x$#*y*#*ys*=*xs*
**by** (*metis create-ldeep-rev.cases list.size*(*3*) *One-nat-def length-Cons less-asym*

*zero-less-Suc*)

**lemma** *hd-in-verts-if-forward'*: ⟦*length xs* > *1*; *forward xs*⟧ ⟹ *hd xs* ∈ *verts T*
**using** *two-elems-if-length-gt1 hd-in-verts-if-forward* **by** *blast*

**lemma** *hd-reach-all-forward'*:
⟦*length xs* > *1*; *forward xs*; *x* ∈ *set xs*⟧ ⟹ *hd xs* →**T x*
**by** (*simp add*: *hd-in-verts-if-forward' hd-reach-all-forward*)

**lemma** *hd-reach-all-forward''*:
⟦*forward* (*x*#*y*#*xs*); *z* ∈ *set* (*x*#*y*#*xs*)⟧ ⟹ *hd* (*x*#*y*#*xs*) →**T z*
**using** *hd-in-verts-if-forward hd-reach-all-forward* **by** *blast*

**lemma** *no-back-if-distinct-forward*: ⟦*forward xs*; *distinct xs*⟧ ⟹ *no-back xs*
**unfolding** *no-back-def* **proof**
**assume** ∃ *i j*. *i* < *j* ∧ *j* < *length xs* ∧ *xs*!*j* →*T xs*!*i* **and** *assms*: *forward xs distinct xs*
**then obtain** *i j* **where** *i-def*: *i* < *j j* < *length xs xs*!*j* →*T xs*!*i* **by** *blast*
**show** *False*
**proof**(*cases i=0*)
**case** *True*
**then have** *xs*!*i* = *hd xs* **using** *i-def*(*1*,*2*) *hd-conv-nth*[*of xs*] **by** *fastforce*
**then have** *xs*!*i* →**T xs*!*j* **using** *i-def*(*1*,*2*) *assms*(*1*) *hd-reach-all-forward'* **by** *simp*
**then have** *xs*!*i* →*+T xs*!*j* **using** *reachable-neq-reachable1 i-def*(*3*) **by** *force*
**then show** *?thesis* **using** *i-def*(*3*) *reachable1-not-reverse* **by** *blast*
**next**
**case** *False*
**then have** *i* ∈ {*1* .. *length xs* − *1*} **using** *i-def*(*1*,*2*) **by** *simp*
**then obtain** *j'* **where** *j'-def*: *j'* < *i xs*!*j'* →*T xs*!*i*
**using** *assms*(*1*) **unfolding** *forward-def* **by** *blast*
**have** *xs*!*j'* = *xs*!*j* **using** *i-def*(*3*) *j'-def*(*2*) *two-in-arcs-contr* **by** *fastforce*
**moreover have** *xs*!*j'* ≠ *xs*!*j*
**using** *j'-def*(*1*) *i-def*(*1*,*2*) *assms*(*2*) *nth-eq-iff-index-eq* **by** *fastforce*
**ultimately show** *?thesis* **by** *blast*
**qed**
**qed**

**corollary** *seq-conform-if-dstnct-fwd*: ⟦*forward xs*; *distinct xs*⟧ ⟹ *seq-conform xs*
**using** *no-back-if-distinct-forward seq-conform-def forward-arcs-alt no-back-arcs-alt*
**by** *blast*

**lemma** *forward-arcs-single*: *forward-arcs* [*x*]
**by** *simp*

**lemma** *forward-single*: *forward* [*x*]
**unfolding** *forward-def* **by** *simp*

**lemma** *no-back-arcs-single*: *no-back-arcs* [*x*]

**by** *simp*

**lemma** *no-back-single*: *no-back* [*x*]
**unfolding** *no-back-def* **by** *simp*

**lemma** *seq-conform-single*: *seq-conform* [*x*]
**unfolding** *seq-conform-def* **by** *simp*

**lemma** *forward-arc-to-head′*:
**assumes** *forward ys* **and** $x \notin$ *set ys* **and** $y \in$ *set ys* **and** $x \rightarrow_T y$
**shows** $y =$ *hd ys*
**proof** (*rule ccontr*)
**assume** *asm*: $y \neq$ *hd ys*
**obtain** *i* **where** *i-def*: $i <$ *length ys* $ys!i = y$ **using** *assms*(*3*) **by** (*auto simp*: *in-set-conv-nth*)
**then have** $i \neq 0$ **using** *asm* **by** (*metis drop0 hd-drop-conv-nth*)
**then have** $i \in \{1..(\textit{length ys} - 1)\}$ **using** *i-def*(*1*) **by** *simp*
**then obtain** *j* **where** *j-def*: $j < i$ $ys!j \rightarrow_T ys!i$
**using** *assms*(*1*) *forward-def* **by** *blast*
**then show** *False* **using** *assms*(*4*,*2*) *j-def*(*2*) *i-def two-in-arcs-contr* **by** *fastforce*
**qed**

**corollary** *forward-arc-to-head*:
⟦*forward ys*; *set xs* ∩ *set ys* = {}; $x \in$ *set xs*; $y \in$ *set ys*; $x \rightarrow_T y$⟧
$\Longrightarrow y =$ *hd ys*
**using** *forward-arc-to-head′* **by** *blast*

**lemma** *forward-app′*:
⟦*forward s1*; *forward s2*; *set s1* ∩ *set s2* = {}; ∃ *x*∈*set s1*. ∃ *y*∈*set s2*. $x \rightarrow_T y$⟧
$\Longrightarrow$ *forward* (*s1*@*s2*)
**using** *forward-app*[*of s1 s2*] *forward-arc-to-head* **by** *blast*

**lemma** *reachable1-from-outside-dom*:
⟦$x \rightarrow^+_T y$; $x \notin$ *set ys*; $y \in$ *set ys*⟧ $\Longrightarrow$ ∃ *x′*. ∃ $y' \in$ *set ys*. $x' \notin$ *set ys* ∧ $x' \rightarrow_T y'$
**by** (*induction x y rule*: *trancl.induct*) *auto*

**lemma** *hd-reachable1-from-outside′*:
⟦$x \rightarrow^+_T y$; *forward ys*; $x \notin$ *set ys*; $y \in$ *set ys*⟧ $\Longrightarrow$ ∃ $y' \in$ *set ys*. $x \rightarrow^+_T$ *hd ys*
**apply**(*induction x y rule*: *trancl.induct*)
**using** *forward-arc-to-head′* **by** *force+*

**lemma** *hd-reachable1-from-outside*:
⟦$x \rightarrow^+_T y$; *forward ys*; *set xs* ∩ *set ys* = {}; $x \in$ *set xs*; $y \in$ *set ys*⟧
$\Longrightarrow$ ∃ $y' \in$ *set ys*. $x \rightarrow^+_T$ *hd ys*
**using** *hd-reachable1-from-outside′* **by** *blast*

**lemma** *reachable1-append-old-if-arc*:
**assumes** ∃ *x*∈*set xs*. ∃ *y*∈*set ys*. $x \rightarrow_T y$
**and** $z \notin$ *set xs*

**and** *forward xs*
**and** *y∈set (xs @ ys)*
**and** *z* $\rightarrow^+{}_T$ *y*
**shows** ∃ *y∈set ys. z* $\rightarrow^+{}_T$ *y*
**proof**(*cases y ∈ set ys*)
**case** *True*
**then show** *?thesis* **using** *assms*(*5*) **by** *blast*
**next**
**case** *False*
**then have** *y ∈ set xs* **using** *assms*(*4*) **by** *simp*
**then have** *0*: *z* $\rightarrow^+{}_T$ *hd xs* **using** *hd-reachable1-from-outside'*[*OF assms*(*5*,*3*,*2*)]
**by** *blast*
**then have** *1*: *hd xs ∈ verts T* **using** *reachable1-in-verts*(*2*) **by** *auto*
**obtain** *x y* **where** *x-def*: *x∈set xs y∈set ys x* $\rightarrow_T$ *y* **using** *assms*(*1*) **by** *blast*
**then have** *hd xs* $\rightarrow^*{}_T$ *x* **using** *hd-reach-all-forward*[*OF 1 assms*(*3*)] **by** *simp*
**then have** *hd xs* $\rightarrow^*{}_T$ *y* **using** *x-def*(*3*) **by** *force*
**then show** *?thesis* **using** *reachable1-reachable-trans*[*OF 0*] *x-def*(*2*) **by** *blast*
**qed**

**lemma** *reachable1-append-old-if-arcU*:
⟦∃ *x∈set xs.* ∃ *y∈set ys. x* $\rightarrow_T$ *y*; *set U* ∩ *set xs* = {}; *z ∈ set U*;
*forward xs*; *y∈set (xs @ ys)*; *z* $\rightarrow^+{}_T$ *y*⟧
⟹ ∃ *y∈set ys. z* $\rightarrow^+{}_T$ *y*
**using** *reachable1-append-old-if-arc*[*of xs ys*] **by** *auto*

**lemma** *before-arc-to-hd*: *before xs ys* ⟹ ∃ *x ∈ set xs. x* $\rightarrow_T$ *hd ys*
**using** *forward-arc-to-head before-def seq-conform-alt* **by** *auto*

**lemma** *no-back-backarc-app1*:
⟦*j < length* (*xs*@*ys*); *j ≥ length xs*; *i < j*; *no-back ys*; (*xs*@*ys*)!*j* $\rightarrow_T$ (*xs*@*ys*)!*i*⟧
⟹ *i < length xs*
**by** (*rule ccontr*) (*auto simp add*: *no-back-def nth-append*)

**lemma** *no-back-backarc-app2*: ⟦*no-back xs*; *i < j*; (*xs*@*ys*)!*j* $\rightarrow_T$ (*xs*@*ys*)!*i*⟧ ⟹ *j* ≥ *length xs*
**by** (*rule ccontr*) (*auto simp add*: *no-back-def nth-append*)

**lemma** *no-back-backarc-i-in-xs*:
⟦*no-back ys*; *j < length* (*xs*@*ys*); *i < j*; (*xs*@*ys*)!*j* $\rightarrow_T$ (*xs*@*ys*)!*i*⟧
⟹ *xs*!*i ∈ set xs* ∧ (*xs*@*ys*)!*i* = *xs*!*i*
**by** (*auto simp add*: *no-back-def nth-append*)

**lemma** *no-back-backarc-j-in-ys*:
⟦*no-back xs*; *j < length* (*xs*@*ys*); *i < j*; (*xs*@*ys*)!*j* $\rightarrow_T$ (*xs*@*ys*)!*i*⟧
⟹ *ys*!(*j*−*length xs*) ∈ *set ys* ∧ (*xs*@*ys*)!*j* = *ys*!(*j*−*length xs*)
**by** (*auto simp add*: *no-back-def nth-append*)

**lemma** *no-back-backarc-difsets*:
**assumes** *no-back xs* **and** *no-back ys*

**and** $i < j$ **and** $j <$ *length* $(xs\ @\ ys)$ **and** $(xs\ @\ ys)\ !\ j \rightarrow_T (xs\ @\ ys)\ !\ i$
**shows** $\exists\, x \in$ *set xs*. $\exists\, y \in$ *set ys*. $y \rightarrow_T x$
**using** *no-back-backarc-i-in-xs*[*OF assms*(*2*,*4*,*3*)] *no-back-backarc-j-in-ys*[*OF assms*(*1*,*4*,*3*)] *assms*(*5*)
**by** *auto*

**lemma** *no-back-backarc-difsets'*:
$[\![$*no-back xs*; *no-back ys*; $\exists\, i\ j.\ i < j \wedge j <$ *length* $(xs@ys) \wedge (xs@ys)!j \rightarrow_T (xs@ys)!i]\!]$
$\Longrightarrow \exists\, x \in$ *set xs*. $\exists\, y \in$ *set ys*. $y \rightarrow_T x$
**using** *no-back-backarc-difsets* **by** *blast*

**lemma** *no-back-before-aux*:
**assumes** *seq-conform xs* **and** *seq-conform ys*
**and** *set xs* $\cap$ *set ys* $= \{\}$ **and** $(\exists\, x{\in}$*set xs*. $\exists\, y{\in}$*set ys*. $x \rightarrow_T y)$
**shows** *no-back* $(xs\ @\ ys)$
**unfolding** *no-back-def* **by** (*metis assms adj-in-verts*(*2*) *forward-arc-to-head hd-reach-all-forward inf-commute reachable1-not-reverse reachable-rtranclI rtrancl-into-trancl1 seq-conform-alt no-back-backarc-difsets'*)

**lemma** *no-back-before*: *before xs ys* $\Longrightarrow$ *no-back* $(xs@ys)$
**using** *before-def no-back-before-aux* **by** *simp*

**lemma** *seq-conform-if-before*: *before xs ys* $\Longrightarrow$ *seq-conform* $(xs@ys)$
**using** *no-back-before before-def seq-conform-alt forward-app before-arc-to-hd* **by** *simp*

**lemma** *no-back-arc-if-fwd-dstct*:
**assumes** *forward* $(as@bs)$ **and** *distinct* $(as@bs)$
**shows** $\neg(\exists\, x{\in}$*set bs*. $\exists\, y{\in}$*set as*. $x \rightarrow_T y)$
**proof**
**assume** $\exists\, x{\in}$*set bs*. $\exists\, y{\in}$*set as*. $x \rightarrow_T y$
**then obtain** *x y* **where** *x-def*: $x{\in}$*set bs* $y{\in}$*set as* $x \rightarrow_T y$ **by** *blast*
**then obtain** *i* **where** *i-def*: $as!i = y\ i <$ *length as* **by** (*auto simp*: *in-set-conv-nth*)
**obtain** *j* **where** *j-def*: $bs!j = x\ j <$ *length bs* **using** *x-def*(*1*) **by** (*auto simp*: *in-set-conv-nth*)
**then have** $(as@bs)!(j+$*length as*$) = x$ **by** (*simp add*: *nth-append*)
**moreover have** $(as@bs)!i = y$ **using** *i-def* **by** (*simp add*: *nth-append*)
**moreover have** $i < (j+$*length as*$)$ **using** *i-def*(*2*) **by** *simp*
**moreover have** $(j+$*length as*$) <$ *length* $(as\ @\ bs)$ **using** *j-def* **by** *simp*
**ultimately show** *False*
**using** *no-back-if-distinct-forward*[*OF assms*] *x-def*(*3*) **unfolding** *no-back-def*
**by** *blast*
**qed**

**lemma** *no-back-reach1-if-fwd-dstct*:
**assumes** *forward* $(as@bs)$ **and** *distinct* $(as@bs)$
**shows** $\neg(\exists\, x{\in}$*set bs*. $\exists\, y{\in}$*set as*. $x \rightarrow^+{}_T y)$
**proof**

**assume** $\exists x{\in}set\ bs.\ \exists y{\in}set\ as.\ x \to^+_T y$
**then obtain** *x y* **where** *x-def*: $x{\in}set\ bs\ y{\in}set\ as\ x \to^+_T y$ **by** *blast*
**have** *fwd-as*: *forward as* **using** *forward-split*[*OF assms*(*1*)] **by** *blast*
**have** *x-as*: $x \notin set\ as$ **using** *x-def*(*1*) *assms*(*2*) **by** *auto*
**show** *False*
**using** *assms*(*1*) *x-def append.assoc list.distinct*(*1*) *Nil-is-append-conv append-Nil2*[*of as@bs*]
*append-eq-append-conv2*[*of as@bs as@bs bs as*] *forward-arc-to-head'* *hd-append2*
*hd-reach-all-forward* *hd-reachable1-from-outside'*[*OF x-def*(*3*) *fwd-as x-as x-def*(*2*)]
*in-set-conv-decomp-first*[*of y as*] *in-set-conv-decomp-last reachable1-from-outside-dom*
*reachable1-in-verts*(*2*) *reachable1-not-reverse reachable1-reachable-trans*
**by** *metis*
**qed**

**lemma** *split-length-i*: $i \le length\ bs \Longrightarrow \exists xs\ ys.\ xs@ys = bs \wedge length\ xs = i$
**using** *length-take append-take-drop-id min-absorb2* **by** *metis*

**lemma** *split-length-i-prefix*:
**assumes** $length\ as \le i\ i < length\ (as@bs)$
**shows** $\exists xs\ ys.\ xs@ys = bs \wedge length\ (as@xs) = i$
**proof** −
**obtain** *n* **where** *n-def*: $n + length\ as = i$
**using** *assms*(*1*) *ab-semigroup-add-class.add.commute le-Suc-ex* **by** *blast*
**then have** $n \le length\ bs$ **using** *assms*(*2*) **by** *simp*
**then show** *?thesis* **using** *split-length-i n-def* **by** *fastforce*
**qed**

**lemma** *forward-alt-aux1*:
**assumes** $i \in \{1..length\ xs - 1\}$ **and** $j{<}i$ **and** $xs!j \to_T xs!i$
**shows** $\exists as\ bs.\ as@bs = xs \wedge length\ as = i \wedge (\exists x \in set\ as.\ x \to_T xs!i)$
**proof** −
**obtain** *as bs* **where** $as@bs = xs \wedge length\ as = i$
**using** *assms*(*1*) *atLeastAtMost-iff diff-le-self le-trans split-length-i*[*of i xs*] **by** *metis*
**then show** *?thesis* **using** *assms*(*2*,*3*) *nth-append*[*of as bs j*] **by** *force*
**qed**

**lemma** *forward-alt-aux1'*:
*forward xs*
$\Longrightarrow \forall i \in \{1..length\ xs - 1\}.\ \exists as\ bs.\ as@bs = xs \wedge length\ as = i \wedge (\exists x \in set\ as.\ x \to_T xs!i)$
**using** *forward-alt-aux1* **unfolding** *forward-def* **by** *fastforce*

**lemma** *forward-alt-aux2*:
$[\![as@bs = xs;\ length\ as = i;\ \exists x \in set\ as.\ x \to_T xs!i]\!] \Longrightarrow \exists j{<}i.\ xs!j \to_T xs!i$
**by** (*auto simp add*: *nth-append in-set-conv-nth*)

**lemma** *forward-alt-aux2'*:

$\forall i \in \{1..length\ xs - 1\}.\ \exists as\ bs.\ as@bs = xs \wedge length\ as = i \wedge (\exists x \in set\ as.\ x \rightarrow_T xs!i)$
$\Longrightarrow$ *forward xs*
**using** *forward-alt-aux2* **unfolding** *forward-def* **by** *blast*

**corollary** *forward-alt*:
$\forall i \in \{1..length\ xs - 1\}.\ \exists as\ bs.\ as@bs = xs \wedge length\ as = i \wedge (\exists x \in set\ as.\ x \rightarrow_T xs!i)$
$\longleftrightarrow$ *forward xs*
**using** *forward-alt-aux1'*[*of xs*] *forward-alt-aux2'* **by** *blast*

**lemma** *move-mid-forward-if-noarc-aux*:
**assumes** $as \neq []$
**and** $\neg(\exists x \in set\ U.\ \exists y \in set\ bs.\ x \rightarrow_T y)$
**and** *forward* $(as@U@bs@cs)$
**and** $i \in \{1..length\ (as@bs@U@cs) - 1\}$
**shows** $\exists j<i.\ (as@bs@U@cs)\ !\ j \rightarrow_T (as@bs@U@cs)\ !\ i$
**proof** −
**have** *0*: $i \in \{1..length\ (as@U@bs@cs) - 1\}$ **using** *assms*(*4*) **by** *auto*
**consider** $i < length\ as \mid i \in \{length\ as..length\ (as@bs) - 1\}$
$\mid i \in \{length\ (as@bs)..length\ (as@bs@U) - 1\}$
$\mid i \geq length\ (as@bs@U)$
**by** *fastforce*
**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then have** $(as@U@bs@cs)!i = (as@bs@U@cs)!i$ **by** (*simp add*: *nth-append*)
**then obtain** *j* **where** *j-def*: $j<i\ (as@U@bs@cs)!j \rightarrow_T ((as@bs)@U@cs)!i$
**using** *assms*(*3*) *0* **unfolding** *forward-def* **by** *fastforce*
**then have** $(as@U@bs@cs)!j = ((as@bs)@U@cs)!j$ **using** *1* **by** (*simp add*: *nth-append*)
**then show** *?thesis* **using** *j-def* **by** *auto*
**next**
**case** *2*
**have** $((as@bs)@U@cs)!i = bs!(i - length\ as)$
**using** *2 assms*(*4*) *nth-append root-in-T directed-tree-axioms in-degree-root-zero*
**by** (*metis directed-tree.in-deg-one-imp-not-root atLeastAtMost-iff diff-diff-cancel diff-is-0-eq diff-le-self diff-less-mono neq0-conv zero-less-diff*)
**then have** *i-in-bs*: $((as@bs)@U@cs)!i \in set\ bs$ **using** *assms*(*4*) *2* **by** *auto*
**have** $(i - length\ as) < length\ bs$ **using** *2 assms*(*4*) **by** *force*
**then have** $((as@bs)@U@cs)!i = (as@U@bs@cs)!(i + length\ U)$
**using** *2* **by** (*auto simp*: *nth-append*)
**moreover have** $(i + length\ U) \in \{1..\ length\ (as@U@bs@cs) - 1\}$ **using** *2 0* **by** *force*
**ultimately obtain** *j* **where** *j-def*:
$j < (i + length\ U)\ (as@U@bs@cs)!j \rightarrow_T ((as@bs)@U@cs)!i$
**using** *assms*(*3*) **unfolding** *forward-def* **by** *fastforce*
**have** $i < length\ (as@bs)$ **using** ‹$i - length\ as < length\ bs$› **by** *force*
**moreover have** $length\ as \leq i$ **using** *2* **by** *simp*

**ultimately obtain** *xs ys* **where** *xs-def*: *bs* = *xs*@*ys length* (*as*@*xs*) = *i*
**using** *split-length-i-prefix* **by** *blast*
**then have** *j* < (*length* (*as*@*U*@*xs*)) **using** *2 j-def*(*1*) **by** *simp*
**then have** (*as*@*U*@*bs*@*cs*)!*j* ∈ *set* (*as*@*U*@*xs*) **by** (*auto simp*: *xs-def*(*1*) *nth-append*)
**then have** (*as*@*U*@*bs*@*cs*)!*j* ∈ *set* (*as*@*xs*) **using** *assms*(*2*) *j-def*(*2*) *i-in-bs* **by** *auto*
**then obtain** *j'* **where** *j'-def*: *j'* < *length* (*as*@*xs*) (*as*@*xs*)!*j'* = (*as*@*U*@*bs*@*cs*)!*j*
**using** *in-set-conv-nth*[*of* (*as*@*U*@*bs*@*cs*)!*j*] *nth-append* **by** *blast*
**then have** ((*as*@*bs*)@*U*@*cs*)!*j'* = (*as*@*U*@*bs*@*cs*)!*j*
**using** *nth-append*[*of as*@*xs*] *xs-def*(*1*) **by** *simp*
**then show** *?thesis* **using** *j-def*(*2*) *j'-def*(*1*) *xs-def*(*2*) **by** *force*
**next**
**case** *3*
**then have** *i-len-U*: *i* − *length* (*as*@*bs*) < *length U* **using** *assms*(*4*) **by** *fastforce*
**have** *i-len-asU*: *i* − *length bs* < *length* (*as*@*U*) **using** *3 assms*(*4*) **by** *force*
**have** ((*as*@*bs*)@*U*@*cs*)!*i* = (*U*@*cs*)!(*i* − *length* (*as*@*bs*))
**using** *3* **by** (*auto simp*: *nth-append*)
**also have** ... = (*as*@*U*)!(*i* − *length bs*)
**using** *3 i-len-U* **by** (*auto simp*: *ab-semigroup-add-class.add.commute nth-append*)
**also have** ... = (*as*@*U*@*bs*@*cs*)!(*i* − *length bs*)
**using** *i-len-asU nth-append*[*of as*@*U*] **by** *simp*
**finally have** *1*: ((*as*@*bs*)@*U*@*cs*)!*i* = (*as*@*U*@*bs*@*cs*)!(*i* − *length bs*) **.**
**have** (*i* − *length bs*) ≥ *length as* **using** *3* **by** *auto*
**then have** (*i* − *length bs*) ≥ *1* **using** *assms*(*1*) *length-0-conv*[*of as*] **by** *force*
**then have** (*i* − *length bs*) ∈ {*1*.. *length* (*as*@*U*@*bs*@*cs*) − *1*} **using** *0* **by** *auto*
**then obtain** *j* **where** *j-def*: *j* < (*i* − *length bs*) (*as*@*U*@*bs*@*cs*)!*j* →$_T$ ((*as*@*bs*)@*U*@*cs*)!*i*
**using** *assms*(*3*) *1* **unfolding** *forward-def* **by** *fastforce*
**have** *length as* ≤ (*i* − *length bs*) **using** *3* **by** *auto*
**then obtain** *xs ys* **where** *xs-def*: *U* = *xs*@*ys length* (*as*@*xs*) = (*i* − *length bs*)
**using** *split-length-i-prefix*[*of as*] *i-len-asU* **by** *blast*
**then have** *j* < (*length* (*as*@*xs*)) **using** *3 j-def*(*1*) **by** *simp*
**then have** (*as*@*U*@*bs*@*cs*)!*j* ∈ *set* (*as*@*bs*@*xs*) **by** (*auto simp*: *xs-def*(*1*) *nth-append*)
**then obtain** *j'* **where** *j'-def*: *j'* < *length* (*as*@*bs*@*xs*) (*as*@*bs*@*xs*)!*j'* = (*as*@*U*@*bs*@*cs*)!*j*
**using** *in-set-conv-nth*[*of* (*as*@*U*@*bs*@*cs*)!*j*] **by** *blast*
**then have** ((*as*@*bs*)@*U*@*cs*)!*j'* = (*as*@*U*@*bs*@*cs*)!*j*
**using** *nth-append*[*of as*@*bs*@*xs*] *xs-def*(*1*) **by** *simp*
**moreover have** *j'* < *i* **using** *j'-def*(*1*) *xs-def*(*2*) *3* **by** *auto*
**ultimately show** *?thesis* **using** *j-def*(*2*) **by** *force*
**next**
**case** *4*
**have** *len-eq*: *length* (*as*@*U*@*bs*) = *length* (*as*@*bs*@*U*) **by** *simp*
**have** ((*as*@*bs*)@*U*@*cs*)!*i* = *cs*!(*i* − *length* (*as*@*bs*@*U*))
**using** *4 nth-append*[*of as*@*bs*@*U*] **by** *simp*
**also have** ... = *cs*!(*i* − *length* (*as*@*U*@*bs*)) **using** *len-eq* **by** *argo*
**finally have** ((*as*@*bs*)@*U*@*cs*)!*i* = ((*as*@*U*@*bs*)@*cs*)!*i* **using** *4 nth-append*[*of as*@*U*@*bs*] **by** *simp*
**then obtain** *j* **where** *j-def*: *j* < *i* (*as*@*U*@*bs*@*cs*)!*j* →$_T$ ((*as*@*bs*)@*U*@*cs*)!*i*

```
    using assms(3) 0 unfolding forward-def by fastforce
   have length (as@U@bs) ≤ i using 4 by auto
   moreover have i < length ((as@U@bs)@cs) using 0 by auto
   ultimately obtain xs ys where xs-def: xs@ys = cs length ((as@U@bs) @ xs)
= i
    using split-length-i-prefix[of as@U@bs i] by blast
   then have j < (length (as@U@bs@xs)) using 4 j-def(1) by simp
  then have (as@U@bs@cs)!j ∈ set (as@bs@U@xs) by (auto simp: xs-def(1)[symmetric]
nth-append)
   then obtain j' where j'-def: j' < length (as@bs@U@xs) (as@bs@U@xs)!j' =
(as@U@bs@cs)!j
    using in-set-conv-nth[of (as@U@bs@cs)!j] by blast
   then have ((as@bs)@U@cs)!j' = (as@U@bs@cs)!j
    using nth-append[of as@bs@U@xs] xs-def(1)[symmetric] by simp
   moreover have j' < i using j'-def(1) xs-def(2) 4 by auto
   ultimately show ?thesis using j-def(2) by auto
 qed
qed

lemma move-mid-forward-if-noarc:
 ⟦as ≠ []; ¬(∃ x ∈ set U. ∃ y ∈ set bs. x →T y); forward (as@U@bs@cs)⟧
   ⟹ forward (as@bs@U@cs)
 using move-mid-forward-if-noarc-aux unfolding forward-def by blast

lemma move-mid-backward-if-noarc-aux:
 assumes ∃ x∈set U. x →T hd V
    and forward V
    and forward (as@U@bs@V@cs)
    and i ∈ {1..length (as@U@V@bs@cs) − 1}
  shows ∃ j<i. (as@U@V@bs@cs) ! j →T (as@U@V@bs@cs) ! i
proof −
 have 0: i ∈ {1..length (as@U@bs@V@cs) − 1} using assms(4) by auto
 consider i < length (as@U) | i = length (as@U) i ≤ length (as@U@V) − 1
   | i ∈ {length (as@U) + 1..length (as@U@V) − 1}
   | i ∈ {length (as@U@V)..length (as@U@V@bs) − 1}
   | i ≥ length (as@U@V@bs)
   by fastforce
 then show ?thesis
 proof(cases)
   case 1
  then have (as@U@bs@V@cs)!i = (as@U@V@bs@cs)!i by (simp add: nth-append)
  then obtain j where j-def: j<i (as@U@bs@V@cs)!j →T (as@U@V@bs@cs)!i
    using assms(3) 0 unfolding forward-def by fastforce
   then have (as@U@V@bs@cs)!j = (as@U@bs@V@cs)!j using 1 by (simp
add: nth-append)
   then show ?thesis using j-def by auto
 next
   case 2
  have (as@U@V@bs@cs)!i = (V@bs@cs)!0 using 2(1) by (auto simp: nth-append)
```

**then have** $(as@U@V@bs@cs)!i = hd\ V$
**using** *2 assms(4) hd-append hd-conv-nth Suc-n-not-le-n atLeastAtMost-iff le-diff-conv2*
**by** (*metis ab-semigroup-add-class.add.commute append.right-neutral Suc-eq-plus1-left*)
**then obtain** *x* **where** *x-def*: $x \in set\ U\ x \rightarrow_T (as@U@V@bs@cs)!i$ **using** *assms(1)* **by** *auto*
**then obtain** *j* **where** *j-def*: $(as@U)!j = x\ j < i$ **using** *in-set-conv-nth*[*of x*] *2* **by** *fastforce*
**then have** $(as@U@V@bs@cs)!j = x$ **using** *2(1)* **by** (*auto simp*: *nth-append*)
**then show** *?thesis* **using** *j-def(2) x-def(2)* **by** *blast*
**next**
**case** *3*
**have** $i - length\ (as@U) \in \{1\ ..\ length\ V - 1\}$ **using** *3* **by** *force*
**then obtain** *j* **where** *j-def*: $j < (i - length\ (as@U))\ V!j \rightarrow_T V!(i - length\ (as@U))$
**using** *assms(2)* **unfolding** *forward-def* **by** *blast*
**then have** $(as@U@V@bs@cs)!(j+length\ (as@U)) = V!j$
**using** *3 nth-append*[*of as@U*] *nth-append*[*of V*] **by** *auto*
**moreover have** $(as@U@V@bs@cs)!i = V!(i - length\ (as@U))$
**using** *3 nth-append*[*of as@U*] *nth-append*[*of V*] **by** *auto*
**moreover have** $j+length\ (as@U) < i$ **using** *j-def(1)* **by** *simp*
**ultimately show** *?thesis* **using** *j-def(2)* **by** *auto*
**next**
**case** *4*
**have** $(as@U@V@bs@cs)!i = (bs@cs)!(i - length\ (as@U@V))$ **using** *4 nth-append*[*of as@U@V*] **by** *simp*
**also have** $\ldots = bs!(i - length\ (as@U@V))$ **using** *4 assms(4)* **by** (*auto simp*: *nth-append*)
**also have** $\ldots = (as@U@bs)!(i - length\ (as@U@V) + length\ (as@U))$ **by** (*simp add*: *nth-append*)
**also have** $\ldots = (as@U@bs)!(i - length\ V)$ **using** *4* **by** *simp*
**finally have** *1*: $(as@U@V@bs@cs)!i = (as@U@bs@V@cs)!(i - length\ V)$
**using** *4 assms(4) nth-append*[*of as@U@bs*] **by** *auto*
**have** $(i - length\ V) \geq length\ (as@U)$ **using** *4* **by** *auto*
**then have** $(i - length\ V) \geq 1$ **using** *assms(1) length-0-conv* **by** *fastforce*
**then have** $(i - length\ V) \in \{1..\ length\ (as@U@bs@V@cs) - 1\}$ **using** *0* **by** *auto*
**then obtain** *j* **where** *j-def*: $j < i - length\ V\ (as@U@bs@V@cs)!j \rightarrow_T (as@U@V@bs@cs)!i$
**using** *assms(3) 1* **unfolding** *forward-def* **by** *fastforce*
**have** $length\ (as@U) \leq (i - length\ V)$ **using** *4* **by** *fastforce*
**moreover have** $(i - length\ V) < length\ ((as@U)@bs)$ **using** *4 assms(4)* **by** *auto*
**ultimately obtain** *xs ys* **where** *xs-def*: $xs@ys = bs\ length\ ((as@U)@\ xs) = i - length\ V$
**using** *split-length-i-prefix*[*of as@U*] **by** *blast*
**then have** $j < (length\ (as@U@xs))$ **using** *4 j-def(1)* **by** *simp*
**then have** $(as@U@bs@V@cs)!j \in set\ (as@U@V@xs)$ **by** (*auto simp*: *xs-def(1)*[*symmetric*] *nth-append*)

**then obtain** $j'$ **where** $j'$*-def*: $j' < length\ (as@U@V@xs)\ (as@U@V@xs)!j' = (as@U@bs@V@cs)!j$
**using** *in-set-conv-nth*[*of* $(as@U@bs@V@cs)!j$] **by** *blast*
**then have** $(as@U@V@bs@cs)!j' = (as@U@bs@V@cs)!j$
**using** *nth-append*[*of* $as@U@V@xs$] *xs-def*(*1*) **by** *auto*
**moreover have** $j' < i$ **using** $j'$*-def*(*1*) *xs-def*(*2*) *4* **by** *auto*
**ultimately show** *?thesis* **using** *j-def*(*2*) **by** *auto*
**next**
**case** *5*
**have** *len-eq*: $length\ (as@U@bs@V) = length\ (as@U@V@bs)$ **by** *simp*
**have** $(as@U@V@bs@cs)!i = cs!(i - length\ (as@U@V@bs))$
**using** *5 nth-append*[*of* $as@U@V@bs$] **by** *auto*
**also have** $\ldots = cs!(i - length\ (as@U@bs@V))$ **using** *len-eq* **by** *argo*
**finally have** $(as@U@V@bs@cs)!i = ((as@U@bs@V)@cs)!i$
**using** *5 nth-append*[*of* $as@U@bs@V$] **by** *simp*
**then obtain** $j$ **where** *j-def*: $j < i\ (as@U@bs@V@cs)!j \rightarrow_T (as@U@V@bs@cs)!i$
**using** *assms*(*3*) *0* **unfolding** *forward-def* **by** *fastforce*
**have** $length\ (as@U@bs@V) \leq i$ **using** *5* **by** *auto*
**moreover have** $i < length\ ((as@U@bs@V)@cs)$ **using** *0* **by** *auto*
**ultimately obtain** *xs ys* **where** *xs-def*: $xs@ys = cs\ length\ ((as@U@bs@V)\ @\ xs) = i$
**using** *split-length-i-prefix*[*of* $as@U@bs@V\ i$] **by** *blast*
**then have** $j < (length\ (as@U@bs@V@xs))$ **using** *5 j-def*(*1*) **by** *simp*
**then have** $(as@U@bs@V@cs)!j \in set\ (as@U@V@bs@xs)$
**by** (*auto simp*: *xs-def*(*1*)[*symmetric*] *nth-append*)
**then obtain** $j'$ **where** $j'$*-def*: $j' < length\ (as@U@V@bs@xs)\ (as@U@V@bs@xs)!j' = (as@U@bs@V@cs)!j$
**using** *in-set-conv-nth*[*of* $(as@U@bs@V@cs)!j$] **by** *blast*
**then have** $(as@U@V@bs@cs)!j' = (as@U@bs@V@cs)!j$
**using** *nth-append*[*of* $as@U@V@bs@xs$] *xs-def*(*1*) **by** *force*
**moreover have** $j' < i$ **using** $j'$*-def*(*1*) *xs-def*(*2*) *5* **by** *auto*
**ultimately show** *?thesis* **using** *j-def*(*2*) **by** *auto*
**qed**
**qed**

**lemma** *move-mid-backward-if-noarc*:
$[\![before\ U\ V;\ forward\ (as@U@bs@V@cs)]\!] \Longrightarrow forward\ (as@U@V@bs@cs)$
**using** *before-forward2I*
**by** (*simp add*: *forward-def before-arc-to-hd move-mid-backward-if-noarc-aux*)

**lemma** *move-mid-backward-if-noarc'*:
$[\![\exists\, x{\in}set\ U.\ \exists\, y{\in}set\ V.\ x \rightarrow_T y;\ forward\ V;\ set\ U \cap set\ V = \{\};\ forward\ (as@U@bs@V@cs)]\!]$
$\Longrightarrow forward\ (as@U@V@bs@cs)$
**using** *move-mid-backward-if-noarc-aux*[*of U V as bs cs*] *forward-arc-to-head*[*of V U*] *forward-def*
**by** *blast*

**end**

## 10.1 Sublist Additions

**lemma** *fst-sublist-if-not-snd-sublist*:
⟦*xs*@*ys*=*A*@*B*; ¬ *sublist B ys*⟧ ⟹ ∃ *as bs*. *as* @ *bs* = *xs* ∧ *bs* @ *ys* = *B*
**by** (*metis suffix-append suffix-def suffix-imp-sublist*)

**lemma** *sublist-before-if-mid*:
**assumes** *sublist U* (*A*@*V*) **and** *A* @ *V* @ *B* = *xs* **and** *set U* ∩ *set V* = {} **and** *U*≠[]
**shows** ∃ *as bs cs*. *as* @ *U* @ *bs* @ *V* @ *cs* = *xs*
**proof** −
**obtain** *C D* **where** *C-def*: (*C* @ *U*) @ *D* = *A* @ *V* **using** *assms*(*1*) **by** (*auto simp*: *sublist-def*)
**have** *sublist V D*
**using** *assms*(*3*,*4*) *fst-sublist-if-not-snd-sublist*[*OF C-def*] *disjoint-iff-not-equal last-appendR*
**by** (*metis Int-iff Un-Int-eq*(*1*) *append-Nil2 append-self-conv2 set-append last-in-set sublist-def*)
**then show** *?thesis* **using** *assms*(*2*) *C-def sublist-def append.assoc* **by** *metis*
**qed**

**lemma** *list-empty-if-subset-dsjnt*: ⟦*set xs* ⊆ *set ys*; *set xs* ∩ *set ys* = {}⟧ ⟹ *xs* = []
**using** *semilattice-inf-class.inf.orderE* **by** *fastforce*

**lemma** *empty-if-sublist-dsjnt*: ⟦*sublist xs ys*; *set xs* ∩ *set ys* = {}⟧ ⟹ *xs* = []
**using** *set-mono-sublist list-empty-if-subset-dsjnt* **by** *fast*

**lemma** *sublist-snd-if-fst-dsjnt*:
**assumes** *sublist U* (*V*@*B*) **and** *set U* ∩ *set V* = {}
**shows** *sublist U B*
**proof** −
**consider** *sublist U V* | *sublist U B* | (∃ *xs1 xs2*. *U* = *xs1*@*xs2* ∧ *suffix xs1 V* ∧ *prefix xs2 B*)
**using** *assms*(*1*) *sublist-append* **by** *blast*
**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then show** *?thesis* **using** *assms*(*2*) *empty-if-sublist-dsjnt* **by** *blast*
**next**
**case** *2*
**then show** *?thesis* **by** *simp*
**next**
**case** *3*
**then obtain** *xs ys* **where** *xs-def*: *U* = *xs*@*ys suffix xs V prefix ys B* **by** *blast*
**then have** *set xs* ⊆ *set V* **by** (*simp add*: *set-mono-suffix*)
**then have** *xs* = [] **using** *xs-def*(*1*) *assms*(*2*) *list-empty-if-subset-dsjnt* **by** *fastforce*
**then show** *?thesis* **using** *xs-def*(*1*,*3*) **by** *simp*
**qed**

**qed**

**lemma** *sublist-fst-if-snd-dsjnt*:
**assumes** *sublist U* (*B*@*V*) **and** *set U* ∩ *set V* = {}
**shows** *sublist U B*
**proof** −
**consider** *sublist U V* | *sublist U B* | (∃ *xs1 xs2*. *U* = *xs1*@*xs2* ∧ *suffix xs1 B* ∧ *prefix xs2 V*)
**using** *assms*(*1*) *sublist-append* **by** *blast*
**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then show** *?thesis* **using** *assms*(*2*) *empty-if-sublist-dsjnt* **by** *blast*
**next**
**case** *2*
**then show** *?thesis* **by** *simp*
**next**
**case** *3*
**then obtain** *xs ys* **where** *xs-def*: *U* = *xs*@*ys suffix xs B prefix ys V* **by** *blast*
**then have** *set ys* ⊆ *set V* **by** (*simp add*: *set-mono-prefix*)
**then have** *ys* = [] **using** *xs-def*(*1*) *assms*(*2*) *list-empty-if-subset-dsjnt* **by** *fastforce*
**then show** *?thesis* **using** *xs-def*(*1*,*2*) **by** *simp*
**qed**
**qed**

**lemma** *sublist-app*: *sublist* (*A* @ *B*) *C* ⟹ *sublist A C* ∧ *sublist B C*
**using** *sublist-order.dual-order.trans* **by** *blast*

**lemma** *sublist-Cons*: *sublist* (*A* # *B*) *C* ⟹ *sublist* [*A*] *C* ∧ *sublist B C*
**using** *sublist-app*[*of* [*A*]] **by** *simp*

**lemma** *sublist-set-elem*: ⟦*sublist xs* (*A*@*B*); *x* ∈ *set xs*⟧ ⟹ *x* ∈ *set A* ∨ *x* ∈ *set B*
**using** *set-mono-sublist* **by** *fastforce*

**lemma** *subset-snd-if-hd-notin-fst*:
**assumes** *sublist ys* (*V* @ *B*) **and** *hd ys* ∉ *set V* **and** *ys* ≠ []
**shows** *set ys* ⊆ *set B*
**proof** −
**have** ¬ *sublist ys V* **using** *assms*(*2*,*3*) **by**(*auto simp*: *sublist-def*)
**then consider** *sublist ys B* | (∃ *xs1 xs2*. *ys* = *xs1*@*xs2* ∧ *suffix xs1 V* ∧ *prefix xs2 B*)
**using** *assms*(*1*) *sublist-append* **by** *blast*
**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then show** *?thesis* **using** *set-mono-sublist* **by** *blast*
**next**
**case** *2*

**then obtain** *xs zs* **where** *xs-def*: *ys* = *xs*@*zs suffix xs V prefix zs B* **by** *blast*
**then have** *set xs* ⊆ *set V* **by** (*simp add*: *set-mono-suffix*)
**then have** *xs* = [] **using** *xs-def*(*1*) *assms*(*2*,*3*) *hd-append hd-in-set subsetD* **by** *fastforce*
**then show** *?thesis* **using** *xs-def*(*1*,*3*) **by** (*simp add*: *set-mono-prefix*)
**qed**
**qed**

**lemma** *suffix-ndjsnt-snd-if-nempty*: ⟦*suffix xs* (*A*@*V*); *V* ≠ []; *xs* ≠ []⟧ ⟹ *set xs* ∩ *set V* ≠ {}
**using** *empty-if-sublist-dsjnt disjoint-iff*
**by** (*metis sublist-append-leftI suffix-append suffix-imp-sublist*)

**lemma** *sublist-not-mid*:
**assumes** *sublist U* ((*A* @ *V*) @ *B*) **and** *set U* ∩ *set V* = {} **and** *V* ≠ []
**shows** *sublist U A* ∨ *sublist U B*
**proof** −
**consider** *sublist U A* | *sublist U V* | (∃ *xs1 xs2*. *U* = *xs1*@*xs2* ∧ *suffix xs1 A* ∧ *prefix xs2 V*)
| *sublist U B* | (∃ *xs1 xs2*. *U* = *xs1*@*xs2* ∧ *suffix xs1* (*A*@*V*) ∧ *prefix xs2 B*)
**using** *assms*(*1*) *sublist-append* **by** *metis*
**then show** *?thesis*
**proof**(*cases*)
**case** *2*
**then show** *?thesis* **using** *assms*(*2*) *empty-if-sublist-dsjnt* **by** *blast*
**next**
**case** *3*
**then show** *?thesis* **using** *assms*(*2*) *sublist-append sublist-fst-if-snd-dsjnt* **by** *blast*
**next**
**case** *5*
**then obtain** *xs ys* **where** *xs-def*: *U* = *xs*@*ys suffix xs* (*A*@*V*) *prefix ys B* **by** *blast*
**then have** *set xs* ∩ *set V* ≠ {} ∨ *xs* = [] **using** *suffix-ndjsnt-snd-if-nempty assms*(*3*) **by** *blast*
**then have** *xs* = [] **using** *xs-def*(*1*) *assms*(*2*) **by** *auto*
**then show** *?thesis* **using** *xs-def*(*1*,*3*) **by** *simp*
**qed**(*auto*)
**qed**

**lemma** *sublist-Y-cases-UV*:
**assumes** ∀ *xs* ∈ *Y*. ∀ *ys* ∈ *Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}
**and** *U* ∈ *Y*
**and** *V* ∈ *Y*
**and** *U* ≠ []
**and** *V* ≠ []
**and** (∀ *xs* ∈ *Y*. *sublist xs* (*as*@*U*@*bs*@*V*@*cs*))
**and** *xs* ∈ *Y*
**shows** *sublist xs as* ∨ *sublist xs bs* ∨ *sublist xs cs* ∨ *U* = *xs* ∨ *V* = *xs*

**using** *assms append-assoc sublist-not-mid* **by** *metis*

**lemma** *sublist-behind-if-nbefore*:
**assumes** *sublist U xs sublist V xs* $\nexists$*as bs cs. as @ U @ bs @ V @ cs = xs set U* $\cap$ *set V* = {}
**shows** $\exists$ *as bs cs. as @ V @ bs @ U @ cs = xs*
**proof** −
**have** *V* ≠ [] **using** *assms*(*1*,*3*) **unfolding** *sublist-def* **by** *blast*
**obtain** *A B* **where** *A-def*: *A @ V @ B = xs* **using** *assms*(*2*) **by** (*auto simp*: *sublist-def*)
**then have** ¬*sublist U A* **unfolding** *sublist-def* **using** *assms*(*3*) **by** *fastforce*
**moreover have** *sublist U* ((*A @ V*) *@ B*) **using** *assms*(*1*) *A-def* **by** *simp*
**ultimately have** *sublist U B* **using** *assms*(*4*) *sublist-not-mid* ‹*V*≠[]› **by** *blast*
**then show** *?thesis* **unfolding** *sublist-def* **using** *A-def* **by** *blast*
**qed**

**lemma** *sublists-preserv-move-U*:
⟦*set xs* ∩ *set U* = {}; *set xs* ∩ *set V* = {}; *V*≠[]; *sublist xs* (*as*@*U*@*bs*@*V*@*cs*)⟧
⟹ *sublist xs* (*as*@*bs*@*U*@*V*@*cs*)
**using** *append-assoc self-append-conv2 sublist-def sublist-not-mid* **by** *metis*

**lemma** *sublists-preserv-move-UY*:
⟦∀ *xs* ∈ *Y*. ∀ *ys* ∈ *Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}; *xs* ∈ *Y*; *U* ∈ *Y*; *V* ∈ *Y*;
*V* ≠ []; *sublist xs* (*as*@*U*@*bs*@*V*@*cs*)⟧
⟹ *sublist xs* (*as*@*bs*@*U*@*V*@*cs*)
**using** *sublists-preserv-move-U append-assoc sublist-appendI* **by** *metis*

**lemma** *sublists-preserv-move-UY-all*:
⟦∀ *xs* ∈ *Y*. ∀ *ys* ∈ *Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}; *U* ∈ *Y*; *V* ∈ *Y*;
*V* ≠ []; ∀ *xs* ∈ *Y*. *sublist xs* (*as*@*U*@*bs*@*V*@*cs*)⟧
⟹ ∀ *xs* ∈ *Y*. *sublist xs* (*as*@*bs*@*U*@*V*@*cs*)
**using** *sublists-preserv-move-UY*[*of Y*] **by** *simp*

**lemma** *sublists-preserv-move-V*:
⟦*set xs* ∩ *set U* = {}; *set xs* ∩ *set V* = {}; *U*≠[]; *sublist xs* (*as*@*U*@*bs*@*V*@*cs*)⟧
⟹ *sublist xs* (*as*@*U*@*V*@*bs*@*cs*)
**using** *append-assoc self-append-conv2 sublist-def sublist-not-mid* **by** *metis*

**lemma** *sublists-preserv-move-VY*:
⟦∀ *xs* ∈ *Y*. ∀ *ys* ∈ *Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}; *xs* ∈ *Y*; *U* ∈ *Y*; *V* ∈ *Y*;
*U* ≠ []; *sublist xs* (*as*@*U*@*bs*@*V*@*cs*)⟧
⟹ *sublist xs* (*as*@*U*@*V*@*bs*@*cs*)
**using** *sublists-preserv-move-V append-assoc sublist-appendI* **by** *metis*

**lemma** *sublists-preserv-move-VY-all*:
⟦∀ *xs* ∈ *Y*. ∀ *ys* ∈ *Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}; *U* ∈ *Y*; *V* ∈ *Y*;
*U* ≠ []; ∀ *xs* ∈ *Y*. *sublist xs* (*as*@*U*@*bs*@*V*@*cs*)⟧
⟹ ∀ *xs* ∈ *Y*. *sublist xs* (*as*@*U*@*V*@*bs*@*cs*)
**using** *sublists-preserv-move-VY*[*of Y*] **by** *simp*

**lemma** *distinct-sublist-first*:
⟦*sublist as* (*x*#*xs*); *distinct* (*x*#*xs*); *x* ∈ *set as*⟧ ⟹ *take* (*length as*) (*x*#*xs*) = *as*
**unfolding** *sublist-def* **using** *distinct-app-trans-l distinct-ys-not-xs hd-in-set*
**by** (*metis list.sel*(*1*) *append-assoc append-eq-conv-conj append-self-conv2 hd-append2*)

**lemma** *distinct-sublist-first-remainder*:
⟦*sublist as* (*x*#*xs*); *distinct* (*x*#*xs*); *x* ∈ *set as*⟧ ⟹ *as* @ *drop* (*length as*) (*x*#*xs*)
= *x*#*xs*
**using** *distinct-sublist-first append-take-drop-id*[*of length as x*#*xs*] **by** *fastforce*

**lemma** *distinct-set-diff*: *distinct* (*xs*@*ys*) ⟹ *set ys* = *set* (*xs*@*ys*) − *set xs*
**by** *auto*

**lemma** *list-of-sublist-concat-eq*:
**assumes** ∀ *as* ∈ *Y*. ∀ *bs* ∈ *Y*. *as* = *bs* ∨ *set as* ∩ *set bs* = {}
**and** ∀ *as* ∈ *Y*. *sublist as xs*
**and** *distinct xs*
**and** *set xs* = ⋃(*set* ‘ *Y*)
**and** *finite Y*
**shows** ∃ *ys*. *set ys* = *Y* ∧ *concat ys* = *xs* ∧ *distinct ys*
**using** *assms* **proof**(*induction Finite-Set.card Y arbitrary*: *Y xs*)
**case** (*Suc n*)
**show** *?case*
**proof**(*cases xs*)
**case** *Nil*
**then have** *Y* = {[]} ∨ *Y* = {} **using** *Suc.prems*(*4*) **by** *auto*
**then have** *set* [[]] = *Y* ∧ *concat* [[]] = *xs* ∧ *distinct* [[]] **using** *Nil Suc.hyps*(*2*)
**by** *auto*
**then show** *?thesis* **by** *blast*
**next**
**case** (*Cons x xs′*)
**then obtain** *as* **where** *as-def*: *x* ∈ *set as as* ∈ *Y* **using** *Suc.prems*(*4*) **by** *auto*
**then have** *0*: *as* @ (*drop* (*length as*) *xs*) = *xs*
**using** *Suc.prems*(*2*,*3*) *distinct-sublist-first-remainder Cons* **by** *fast*
**then have** ∀ *bs* ∈ (*Y* − {*as*}). *sublist bs* (*drop* (*length as*) *xs*)
**using** *Suc.prems*(*1*,*2*) *as-def*(*2*) **by** (*metis DiffE insertI1 sublist-snd-if-fst-dsjnt*)
**moreover have** ∀ *cs* ∈ (*Y* − {*as*}). ∀ *bs* ∈ (*Y* − {*as*}). *cs* = *bs* ∨ *set cs* ∩ *set*
*bs* = {}
**using** *Suc.prems*(*1*) **by** *simp*
**moreover have** *distinct* (*drop* (*length as*) *xs*) **using** *Suc.prems*(*3*) **by** *simp*
**moreover have** *set* (*drop* (*length as*) *xs*) = ⋃ (*set* ‘ (*Y*−{*as*}))
**using** *Suc.prems*(*1*,*3*,*4*) *distinct-set-diff*[*of as drop* (*length as*) *xs*] *as-def*(*2*)
*0* **by** *auto*
**moreover have** *n* = *Finite-Set.card* (*Y*−{*as*}) **using** *Suc.hyps*(*2*) *as-def*(*2*)
*Suc.prems*(*5*) **by** *simp*
**ultimately obtain** *ys* **where** *ys-def*:
*set ys* = (*Y*−{*as*}) *concat ys* = *drop* (*length as*) *xs distinct ys*
**using** *Suc.hyps*(*1*) *Suc.prems*(*5*) **by** *blast*

**then have** *set* (*as*#*ys*) = *Y* ∧ *concat* (*as*#*ys*) = *xs* ∧ *distinct* (*as*#*ys*) **using** *0 as-def*(*2*) **by** *auto*
**then show** *?thesis* **by** *blast*
**qed**
**qed**(*auto*)

**lemma** *extract-length-decr*[*termination-simp*]:
*List.extract P xs* = *Some* (*as*,*x*,*bs*) ⟹ *length bs* < *length xs*
**by** (*simp add*: *extract-Some-iff*)

**fun** *separate-P* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list* ⇒ '*a list* × '*a list* **where**
*separate-P P acc xs* = (*case List.extract P xs of*
*None* ⇒ (*acc*,*xs*)
| *Some* (*as*,*x*,*bs*) ⇒ (*case separate-P P* (*x*#*acc*) *bs of* (*acc*',*xs*') ⇒ (*acc*', *as*@*xs*')))

**lemma** *separate-not-P-snd*: *separate-P P acc xs* = (*as*,*bs*) ⟹ ∀ *x* ∈ *set bs*. ¬*P x*
**proof**(*induction P acc xs arbitrary*: *as bs rule*: *separate-P.induct*)
**case** (*1 P acc xs*)
**then show** *?case*
**proof**(*cases List.extract P xs*)
**case** *None*
**then have** *bs* = *xs* **using** *1.prems* **by** *simp*
**then show** *?thesis* **using** *None* **by** (*simp add*: *extract-None-iff*)
**next**
**case** (*Some a*)
**then obtain** *cs x ds* **where** *x-def*[*simp*]: *a* = (*cs*,*x*,*ds*) **by**(*cases a*) *auto*
**then obtain** *acc*' *xs*' **where** *acc*'*-def*: *separate-P P* (*x*#*acc*) *ds* = (*acc*',*xs*') **by** *fastforce*
**then have** (*acc*', *cs*@*xs*') = (*as*,*bs*) **using** *1.prems Some* **by** *simp*
**moreover have** ∀ *x* ∈ *set xs*'. ¬*P x* **using** *1.IH acc*'*-def Some x-def* **by** *blast*
**ultimately show** *?thesis* **using** *Some* **by** (*auto simp*: *extract-Some-iff*)
**qed**
**qed**

**lemma** *separate-input-impl-none*: *separate-P P acc xs* = (*acc*,*xs*) ⟹ *List.extract P xs* = *None*
**using** *extract-None-iff separate-not-P-snd* **by** *fast*

**lemma** *separate-input-iff-none*: *List.extract P xs* = *None* ⟷ *separate-P P acc xs* = (*acc*,*xs*)
**using** *separate-input-impl-none* **by** *auto*

**lemma** *separate-P-fst-acc*:
*separate-P P acc xs* = (*as*,*bs*) ⟹ ∃ *as*'. *as* = *as*'@*acc* ∧ (∀ *x* ∈ *set as*'. *P x*)
**proof**(*induction P acc xs arbitrary*: *as bs rule*: *separate-P.induct*)
**case** (*1 P acc xs*)
**then show** *?case*
**proof**(*cases List.extract P xs*)

**case** *None*
**then show** *?thesis* **using** *1.prems* **by** *simp*
**next**
**case** (*Some a*)
**then obtain** *cs x ds* **where** *x-def*[*simp*]: *a* = (*cs*,*x*,*ds*) **by**(*cases a*) *auto*
**then obtain** *acc′ xs′* **where** *acc′-def*: *separate-P P* (*x*#*acc*) *ds* = (*acc′*,*xs′*) **by** *fastforce*
**then have** (*acc′*, *cs*@*xs′*) = (*as*,*bs*) **using** *1.prems Some* **by** *simp*
**then have** ∃ *as′*. *as* = *as′*@(*x*#*acc*) ∧ (∀ *x* ∈ *set as′*. *P x*)
**using** *1.IH acc′-def Some x-def* **by** *blast*
**then show** *?thesis* **using** *Some* **by** (*auto simp*: *extract-Some-iff*)
**qed**
**qed**

**lemma** *separate-P-fst*: *separate-P P* [] *xs* = (*as*,*bs*) ⟹ ∀ *x* ∈ *set as*. *P x*
**using** *separate-P-fst-acc* **by** *fastforce*

## 10.2 Optimal Solution for Lists of Fixed Sets

**lemma** *distinct-seteq-set-length-eq*:
*x* ∈ {*ys*. *set ys* = *xs* ∧ *distinct ys*} ⟹ *length x* = *Finite-Set.card xs*
**using** *distinct-card* **by** *fastforce*

**lemma** *distinct-seteq-set-Cons*:
⟦*Finite-Set.card xs* = *Suc n*; *x* ∈ {*ys*. *set ys* = *xs* ∧ *distinct ys*}⟧
⟹ ∃ *y ys*. *y* # *ys* = *x* ∧ *length ys* = *n* ∧ *distinct ys* ∧ *finite* (*set ys*)
**using** *distinct-seteq-set-length-eq*[*of x*] *Suc-length-conv*[*of n x*] **by** *force*

**lemma** *distinct-seteq-set-Cons′*:
⟦*Finite-Set.card xs* = *Suc n*; *x* ∈ {*ys*. *set ys* = *xs* ∧ *distinct ys*}⟧
⟹ ∃ *y ys zs*. *y* # *ys* = *x* ∧ *Finite-Set.card zs* = *n* ∧ *distinct ys* ∧ *set ys* = *zs*
**using** *distinct-seteq-set-length-eq*[*of x*] *Suc-length-conv*[*of n x*] **by** *force*

**lemma** *distinct-seteq-set-Cons″*:
⟦*Finite-Set.card xs* = *Suc n*; *x* ∈ {*ys*. *set ys* = *xs* ∧ *distinct ys*}⟧
⟹ ∃ *y ys zs*. *y* # *ys* = *x* ∧ *y* ∈ *xs*
∧ *set ys* = *zs* ∧ *Finite-Set.card zs* = *n* ∧ *distinct ys* ∧ *finite zs*
**using** *distinct-seteq-set-Cons* **by** *fastforce*

**lemma** *distinct-seteq-set-Cons-in-set*:
⟦*Finite-Set.card xs* = *Suc n*; *x* ∈ {*ys*. *set ys* = *xs* ∧ *distinct ys*}⟧
⟹ ∃ *y ys zs*. *y*#*ys* = *x* ∧ *y* ∈ *xs* ∧ *Finite-Set.card zs* = *n* ∧ *ys*∈{*ys*. *set ys* = *zs* ∧ *distinct ys*}
**using** *distinct-seteq-set-Cons″* **by** *auto*

**lemma** *distinct-seteq-set-Cons-in-set′*:
⟦*Finite-Set.card xs* = *Suc n*; *x* ∈ {*ys*. *set ys* = *xs* ∧ *distinct ys*}⟧
⟹ ∃ *y ys*. *x* = *y*#*ys* ∧ *y* ∈ *xs* ∧ *ys*∈{*ys*. *set ys* = (*xs* − {*y*}) ∧ *distinct ys*}
**using** *distinct-seteq-set-Cons″* **by** *fastforce*

**lemma** *distinct-seteq-eq-set-union*:
*Finite-Set.card xs = Suc n*
$\Longrightarrow$ $\{ys.\ set\ ys = xs \wedge distinct\ ys\}$
$= \{y\ \#\ ys\ |y\ ys.\ y \in xs \wedge ys \in \{as.\ set\ as = (xs - \{y\}) \wedge distinct\ as\}\}$
**using** *distinct-seteq-set-Cons-in-set'* **by** *force*

**lemma** *distinct-seteq-sub-set-union*:
*Finite-Set.card xs = Suc n*
$\Longrightarrow$ $\{ys.\ set\ ys = xs \wedge distinct\ ys\}$
$\subseteq \{y\ \#\ ys\ |y\ ys.\ y \in xs \wedge ys \in \{as.\ \exists\, a \in xs.\ set\ as = (xs - \{a\}) \wedge distinct\ as\}\}$
**using** *distinct-seteq-set-Cons-in-set'* **by** *fast*

**lemma** *finite-set-union*: $[\![finite\ ys;\ \forall\, y \in ys.\ finite\ y]\!] \Longrightarrow finite\ (\bigcup y \in ys.\ y)$
**by** *simp*

**lemma** *Cons-set-eq-union-set*:
$\{x\ \#\ y \mid x\ y\ y'.\ x \in xs \wedge y \in y' \wedge y' \in ys\} = \{x\ \#\ y \mid x\ y.\ x \in xs \wedge y \in (\bigcup y \in ys.\ y)\}$
**by** *blast*

**lemma** *finite-set-Cons-union-finite*:
$[\![finite\ xs;\ finite\ ys;\ \forall\, y \in ys.\ finite\ y]\!]$
$\Longrightarrow finite\ \{x\ \#\ y \mid x\ y.\ x \in xs \wedge y \in (\bigcup y \in ys.\ y)\}$
**by** (*simp add*: *finite-image-set2*)

**lemma** *finite-set-Cons-finite*:
$[\![finite\ xs;\ finite\ ys;\ \forall\, y \in ys.\ finite\ y]\!]$
$\Longrightarrow finite\ \{x\ \#\ y \mid x\ y\ y'.\ x \in xs \wedge y \in y' \wedge y' \in ys\}$
**using** *Cons-set-eq-union-set*[*of xs*] **by** (*simp add*: *finite-image-set2*)

**lemma** *finite-set-Cons-finite'*:
$[\![finite\ xs;\ finite\ ys]\!] \Longrightarrow finite\ \{x\ \#\ y\ |x\ y.\ x \in xs \wedge y \in ys\}$
**by** (*auto simp add*: *finite-image-set2*)

**lemma** *Cons-set-alt*: $\{x\ \#\ y\ |x\ y.\ x \in xs \wedge y \in ys\} = \{zs.\ \exists\, x\ y.\ x\ \#\ y = zs \wedge x \in xs \wedge y \in ys\}$
**by** *blast*

**lemma** *Cons-set-sub*:
**assumes** *Finite-Set.card xs = Suc n*
**shows** $\{ys.\ set\ ys = xs \wedge distinct\ ys\}$
$\subseteq \{x\ \#\ y\ |x\ y.\ x \in xs \wedge y \in (\bigcup y \in xs.\ \{as.\ set\ as = xs - \{y\} \wedge distinct\ as\})\}$
**using** *distinct-seteq-eq-set-union*[*OF assms*] **by** *auto*

**lemma** *distinct-seteq-finite*: *finite xs* $\Longrightarrow$ *finite* $\{ys.\ set\ ys = xs \wedge distinct\ ys\}$
**by**(*blast intro*: *rev-finite-subset*[*OF finite-subset-distinct*])

**lemma** *distinct-setsub-split*:
$\{ys.\ set\ ys \subseteq xs \wedge distinct\ ys\}$
$= \{ys.\ set\ ys = xs \wedge distinct\ ys\} \cup (\bigcup y \in xs.\ \{ys.\ set\ ys \subseteq (xs - \{y\}) \wedge distinct\ ys\})$
**by** *blast*

**lemma** *valid-UV-lists-finite*:
$finite\ xs \Longrightarrow finite\ \{x.\ \exists as\ bs\ cs.\ as @ U @ bs @ V @ cs = x \wedge set\ x = xs \wedge distinct\ x\}$
**using** *distinct-seteq-finite* **by** *force*

**lemma** *valid-UV-lists-r-subset*:
$\{x.\ \exists as\ bs\ cs.\ as @ U @ bs @ V @ cs = x \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r]\}$
$\subseteq \{x.\ \exists as\ bs\ cs.\ as @ U @ bs @ V @ cs = x \wedge set\ x = xs \wedge distinct\ x\}$
**by** *blast*

**lemma** *valid-UV-lists-r-finite*:
$finite\ xs \Longrightarrow finite\ \{x.\ \exists as\ bs\ cs.\ as @ U @ bs @ V @ cs = x \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r]\}$
**using** *valid-UV-lists-finite finite-subset*[*OF valid-UV-lists-r-subset*] **by** *fast*

**lemma** *valid-UV-lists-arg-min-ex-aux*:
$[\![finite\ ys;\ ys \neq \{\};\ ys = \{x.\ \exists as\ bs\ cs.\ as @ U @ bs @ V @ cs = x \wedge set\ x = xs \wedge distinct\ x\}]\!]$
$\Longrightarrow \exists y \in ys.\ \forall z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
**using** *arg-min-if-finite*(*1*)[*of ys f*] *arg-min-least*[*of ys*, **where** *?f = f*] **by** *auto*

**lemma** *valid-UV-lists-arg-min-ex*:
$[\![finite\ xs;\ ys \neq \{\};\ ys = \{x.\ \exists as\ bs\ cs.\ as @ U @ bs @ V @ cs = x \wedge set\ x = xs \wedge distinct\ x\}]\!]$
$\Longrightarrow \exists y \in ys.\ \forall z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
**using** *valid-UV-lists-finite valid-UV-lists-arg-min-ex-aux*[*of ys*] **by** *blast*

**lemma** *valid-UV-lists-arg-min-r-ex-aux*:
$[\![finite\ ys;\ ys \neq \{\};$
$ys = \{x.\ \exists as\ bs\ cs.\ as @ U @ bs @ V @ cs = x \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r]\}]\!]$
$\Longrightarrow \exists y \in ys.\ \forall z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
**using** *arg-min-if-finite*(*1*)[*of ys f*] *arg-min-least*[*of ys*, **where** *?f = f*] **by** *auto*

**lemma** *valid-UV-lists-arg-min-r-ex*:
$[\![finite\ xs;\ ys \neq \{\};$
$ys = \{x.\ \exists as\ bs\ cs.\ as @ U @ bs @ V @ cs = x \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r]\}]\!]$
$\Longrightarrow \exists y \in ys.\ \forall z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
**using** *valid-UV-lists-r-finite*[*of xs*] *valid-UV-lists-arg-min-r-ex-aux*[*of ys*] **by** *blast*

**lemma** *valid-UV-lists-nemtpy*:
**assumes** *finite xs set* $(U @ V) \subseteq xs$ *distinct* $(U @ V)$

**shows** $\{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x\} \neq \{\}$
**proof** −
**obtain** *cs* **where** *set cs* = *xs* − *set* (*U*@*V*) ∧ *distinct cs*
**using** *assms*(*1*) *finite-distinct-list*[*of xs* − *set* (*U*@*V*)] **by** *blast*
**then have** []@*U*@[]@*V*@*cs* = *U*@*V*@*cs set* (*U*@*V*@*cs*) = *xs distinct* (*U*@*V*@*cs*)
**using** *assms* **by** *auto*
**then show** *?thesis* **by** *blast*
**qed**

**lemma** *valid-UV-lists-nemtpy′*:
⟦*finite xs*; *set U* ∩ *set V* = {}; *set U* ⊆ *xs*; *set V* ⊆ *xs*; *distinct U*; *distinct V*⟧
⟹ $\{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x\} \neq \{\}$
**using** *valid-UV-lists-nemtpy*[*of xs*] **by** *simp*

**lemma** *valid-UV-lists-nemtpy-r*:
**assumes** *finite xs* **and** *set* (*U*@*V*) ⊆ *xs* **and** *distinct* (*U*@*V*)
**and** *take 1 U* = [*r*] ∨ *r* ∉ *set U* ∪ *set V* **and** *r* ∈ *xs*
**shows** $\{x.\ (\exists\, as\ bs\ cs.\ as@U@bs@V@cs = x) \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r]\} \neq \{\}$
**proof**(*cases take 1 U* = [*r*])
**case** *True*
**obtain** *cs* **where** *set cs* = *xs* − *set* (*U*@*V*) ∧ *distinct cs*
**using** *assms*(*1*) *finite-distinct-list* **by** *auto*
**then have** []@*U*@[]@*V*@*cs* = *U*@*V*@*cs set* (*U*@*V*@*cs*) = *xs distinct* (*U*@*V*@*cs*)
**using** *assms* **by** *auto*
**then show** *?thesis* **using** *True take1-singleton-app* **by** *fast*
**next**
**case** *False*
**obtain** *cs* **where** *cs-def*: *set cs* = *xs* − ({*r*} ∪ *set* (*U*@*V*)) ∧ *distinct cs*
**using** *assms*(*1*) *finite-distinct-list* **by** *auto*
**then have** [*r*]@*U*@[]@*V*@*cs* = [*r*]@*U*@*V*@*cs set* ([*r*]@*U*@*V*@*cs*) = *xs distinct* ([*r*]@*U*@*V*@*cs*)
*take 1* ([*r*]@*U*@*V*@*cs*) = [*r*]
**using** *assms False* **by** *auto*
**then show** *?thesis* **by** (*smt* (*verit*, *del-insts*) *empty-Collect-eq*)
**qed**

**lemma** *valid-UV-lists-nemtpy-r′*:
⟦*finite xs*; *set U* ∩ *set V* = {}; *set U* ⊆ *xs*; *set V* ⊆ *xs*; *distinct U*; *distinct V*;
*take 1 U* = [*r*] ∨ *r* ∉ *set U* ∪ *set V*; *r* ∈ *xs*⟧
⟹ $\{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r]\} \neq \{\}$
**using** *valid-UV-lists-nemtpy-r*[*of xs*] **by** *simp*

**lemma** *valid-UV-lists-arg-min-ex′*:
⟦*finite xs*; *set U* ∩ *set V* = {}; *set U* ⊆ *xs*; *set V* ⊆ *xs*; *distinct U*; *distinct V*;
*ys* = $\{x.\ (\exists\, as\ bs\ cs.\ as@U@bs@V@cs = x) \wedge set\ x = xs \wedge distinct\ x\}$⟧
⟹ $\exists\, y \in ys.\ \forall\, z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
**using** *valid-UV-lists-arg-min-ex*[*of xs*] *valid-UV-lists-nemtpy′*[*of xs*] **by** *simp*

**lemma** *valid-UV-lists-arg-min-r-ex'*:
⟦*finite xs*; *set U* ∩ *set V* = {}; *set U* ⊆ *xs*; *set V* ⊆ *xs*; *distinct U*; *distinct V*;
*take 1 U* = [*r*] ∨ *r* ∉ *set U* ∪ *set V*; *r* ∈ *xs*;
*ys* = {*x*. (∃ *as bs cs*. *as*@*U*@*bs*@*V*@*cs* = *x*) ∧ *set x* = *xs* ∧ *distinct x* ∧ *take 1 x* = [*r*]}⟧
⟹ ∃ *y* ∈ *ys*. ∀ *z* ∈ *ys*. (*f* :: *'a list* ⇒ *real*) *y* ≤ *f z*
**using** *valid-UV-lists-arg-min-r-ex*[*of xs*] *valid-UV-lists-nemtpy-r'*[*of xs*] **by** *simp*

**lemma** *valid-UV-lists-alt*:
**assumes** *P* = (λ*x*. (∃ *as bs cs*. *as*@*U*@*bs*@*V*@*cs* = *x*) ∧ *set x* = *xs* ∧ *distinct x*)
**shows** {*x*. (∃ *as bs cs*. *as*@*U*@*bs*@*V*@*cs* = *x*) ∧ *set x* = *xs* ∧ *distinct x*} = {*ys*. *P ys*}
**using** *assms* **by** *simp*

**lemma** *valid-UV-lists-argmin-ex*:
**fixes** *cost* :: *'a list* ⇒ *real*
**assumes** *P* = (λ*x*. (∃ *as bs cs*. *as*@*U*@*bs*@*V*@*cs* = *x*) ∧ *set x* = *xs* ∧ *distinct x*)
**and** *finite xs*
**and** *set U* ∩ *set V* = {}
**and** *set U* ⊆ *xs*
**and** *set V* ⊆ *xs*
**and** *distinct U*
**and** *distinct V*
**shows** ∃ *as' bs' cs'*. *P* (*as'*@*U*@*bs'*@*V*@*cs'*) ∧
(∀ *as bs cs*. *P* (*as*@*U*@*bs*@*V*@*cs*) ⟶ *cost* (*as'*@*U*@*bs'*@*V*@*cs'*) ≤ *cost* (*as*@*U*@*bs*@*V*@*cs*))
**proof** −
**obtain** *y* **where** *y* ∈ {*ys*. *P ys*} ∧ (∀ *z* ∈ {*ys*. *P ys*}. *cost y* ≤ *cost z*)
**using** *valid-UV-lists-arg-min-ex'*[*OF assms*(2−7)] *assms*(1) **by** *fastforce*
**then show** *?thesis* **using** *assms*(1) **by** *blast*
**qed**

**lemma** *valid-UV-lists-argmin-ex-noP*:
**fixes** *cost* :: *'a list* ⇒ *real*
**assumes** *finite xs*
**and** *set U* ∩ *set V* = {}
**and** *set U* ⊆ *xs*
**and** *set V* ⊆ *xs*
**and** *distinct U*
**and** *distinct V*
**shows** ∃ *as' bs' cs'*. *set* (*as'* @ *U* @ *bs'* @ *V* @ *cs'*) = *xs* ∧ *distinct* (*as'* @ *U* @ *bs'* @ *V* @ *cs'*)
∧ (∀ *as bs cs*. *set* (*as* @ *U* @ *bs* @ *V* @ *cs*) = *xs* ∧ *distinct* (*as* @ *U* @ *bs* @ *V* @ *cs*)
⟶ *cost* (*as'* @ *U* @ *bs'* @ *V* @ *cs'*) ≤ *cost* (*as* @ *U* @ *bs* @ *V* @ *cs*))
**using** *valid-UV-lists-argmin-ex*[*OF refl assms*] **by** *metis*

**lemma** *valid-UV-lists-argmin-r-ex*:

**fixes** *cost* :: *'a list* ⇒ *real*
**assumes** $P = (\lambda x.\ (\exists\, as\ bs\ cs.\ as@U@bs@V@cs = x) \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r])$
**and** *finite xs*
**and** $set\ U \cap set\ V = \{\}$
**and** $set\ U \subseteq xs$
**and** $set\ V \subseteq xs$
**and** *distinct U*
**and** *distinct V*
**and** $take\ 1\ U = [r] \vee r \notin set\ U \cup set\ V$
**and** $r \in xs$
**shows** $\exists\, as'\ bs'\ cs'.\ P\ (as'@U@bs'@V@cs') \wedge$
$(\forall\, as\ bs\ cs.\ P\ (as@U@bs@V@cs) \longrightarrow cost\ (as'@U@bs'@V@cs') \leq cost\ (as@U@bs@V@cs))$
**proof** −
**obtain** *y* **where** $y \in \{ys.\ P\ ys\} \wedge (\forall\, z \in \{ys.\ P\ ys\}.\ cost\ y \leq cost\ z)$
**using** *valid-UV-lists-arg-min-r-ex'*[*OF assms*(2−9)] *assms*(1) **by** *fastforce*
**then show** *?thesis* **using** *assms*(1) **by** *blast*
**qed**

**lemma** *valid-UV-lists-argmin-r-ex-noP*:
**fixes** *cost* :: *'a list* ⇒ *real*
**assumes** *finite xs*
**and** $set\ U \cap set\ V = \{\}$
**and** $set\ U \subseteq xs$
**and** $set\ V \subseteq xs$
**and** *distinct U*
**and** *distinct V*
**and** $take\ 1\ U = [r] \vee r \notin set\ U \cup set\ V$
**and** $r \in xs$
**shows** $\exists\, as'\ bs'\ cs'.\ set\ (as' @ U @ bs' @ V @ cs') = xs$
$\wedge\ distinct\ (as' @ U @ bs' @ V @ cs') \wedge take\ 1\ (as' @ U @ bs' @ V @ cs') = [r]$
$\wedge\ (\forall\, as\ bs\ cs.\ set\ (as @ U @ bs @ V @ cs) = xs$
$\wedge\ distinct\ (as @ U @ bs @ V @ cs) \wedge take\ 1\ (as @ U @ bs @ V @ cs) = [r]$
$\longrightarrow cost\ (as' @ U @ bs' @ V @ cs') \leq cost\ (as @ U @ bs @ V @ cs))$
**using** *valid-UV-lists-argmin-r-ex*[*OF refl assms*] **by** *metis*

**lemma** *valid-UV-lists-argmin-r-ex-noP'*:
**fixes** *cost* :: *'a list* ⇒ *real*
**assumes** *finite xs*
**and** $set\ U \cap set\ V = \{\}$
**and** $set\ U \subseteq xs$
**and** $set\ V \subseteq xs$
**and** *distinct U*
**and** *distinct V*
**and** $take\ 1\ U = [r] \vee r \notin set\ U \cup set\ V$
**and** $r \in xs$
**shows** $\exists\, as'\ bs'\ cs'.\ set\ (as' @ U @ bs' @ V @ cs') = xs$

$\wedge$ *distinct* (*as'* @ *U* @ *bs'* @ *V* @ *cs'*) $\wedge$ *take 1* (*as'* @ *U* @ *bs'* @ *V* @ *cs'*) = [*r*]
$\wedge$ ($\forall$ *as bs cs*. *set* (*as* @ *U* @ *bs* @ *V* @ *cs*) = *xs*
$\wedge$ *distinct* (*as* @ *U* @ *bs* @ *V* @ *cs*) $\wedge$ *take 1* (*as* @ *U* @ *bs* @ *V* @ *cs*) = [*r*]
$\longrightarrow$ *cost* (*rev* (*as'* @ *U* @ *bs'* @ *V* @ *cs'*)) $\leq$ *cost* (*rev* (*as* @ *U* @ *bs* @ *V* @ *cs*)))
**using** *valid-UV-lists-argmin-r-ex-noP*[*OF assms*] **by** *meson*

**lemma** *take1-split-nempty*: *ys* $\neq$ [] $\Longrightarrow$ *take 1* (*xs*@*ys*@*zs*) = *take 1* (*xs*@*ys*)
**by** (*metis append.assoc append-Nil2 gr-zeroI length-0-conv less-one same-append-eq take-append take-eq-Nil zero-less-diff*)

**lemma** *take1-elem*: ⟦*take 1* (*xs*@*ys*) = [*r*]; *r* $\in$ *set xs*⟧ $\Longrightarrow$ *take 1 xs* = [*r*]
**using** *in-set-conv-decomp-last*[*of r xs*] **by** *auto*

**lemma** *take1-nelem*: ⟦*take 1* (*xs*@*ys*) = [*r*]; *r* $\notin$ *set ys*⟧ $\Longrightarrow$ *take 1 xs* = [*r*]
**using** *take1-elem*[*of xs ys r*] *append-self-conv2*[*of xs*] *hd-in-set*[*of ys*]
**by** (*fastforce dest*: *hd-eq-take1*)

**lemma** *take1-split-nelem-nempty*: ⟦*take 1* (*xs*@*ys*@*zs*) = [*r*]; *ys* $\neq$ []; *r* $\notin$ *set ys*⟧
$\Longrightarrow$ *take 1 xs* = [*r*]
**using** *take1-split-nempty take1-nelem* **by** *fastforce*

**lemma** *take1-empty-if-nelem*: ⟦*take 1* (*as*@*bs*@*cs*) = [*r*]; *r* $\notin$ *set as*⟧ $\Longrightarrow$ *as* = []
**using** *take1-split-nelem-nempty*[*of* [] *as bs*@*cs*] **by** *auto*

**lemma** *take1-empty-if-mid*: ⟦*take 1* (*as*@*bs*@*cs*) = [*r*]; *r* $\in$ *set bs*; *distinct* (*as*@*bs*@*cs*)⟧
$\Longrightarrow$ *as* = []
**using** *take1-empty-if-nelem* **by** *fastforce*

**lemma** *take1-mid-if-elem*:
⟦*take 1* (*as*@*bs*@*cs*) = [*r*]; *r* $\in$ *set bs*; *distinct* (*as*@*bs*@*cs*)⟧ $\Longrightarrow$ *take 1 bs* = [*r*]
**using** *take1-empty-if-mid*[*of as bs cs*] **by** (*fastforce intro*: *take1-elem*)

**lemma** *contr-optimal-nogap-no-r*:
**assumes** *asi rank r cost*
**and** *rank* (*rev V*) $\leq$ *rank* (*rev U*)
**and** *finite xs*
**and** *set U* $\cap$ *set V* = {}
**and** *set U* $\subseteq$ *xs*
**and** *set V* $\subseteq$ *xs*
**and** *distinct U*
**and** *distinct V*
**and** *r* $\notin$ *set U* $\cup$ *set V*
**and** *r* $\in$ *xs*
**shows** $\exists$ *as' cs'*. *distinct* (*as'* @ *U* @ *V* @ *cs'*) $\wedge$ *take 1* (*as'* @ *U* @ *V* @ *cs'*) = [*r*]
$\wedge$ *set* (*as'* @ *U* @ *V* @ *cs'*) = *xs* $\wedge$ ($\forall$ *as bs cs*. *set* (*as* @ *U* @ *bs* @ *V* @ *cs*) = *xs*

$\wedge$ *distinct* (*as* @ *U* @ *bs* @ *V* @ *cs*) $\wedge$ *take 1* (*as* @ *U* @ *bs* @ *V* @ *cs*) = [*r*]
$\longrightarrow$ *cost* (*rev* (*as'* @ *U* @ *V* @ *cs'*)) $\leq$ *cost* (*rev* (*as* @ *U* @ *bs* @ *V* @ *cs*)))
**proof** −
**define** *P* **where** *P ys* $\equiv$ *set ys* = *xs* $\wedge$ *distinct ys* $\wedge$ *take 1 ys* = [*r*] **for** *ys*
**obtain** *as' bs' cs'* **where** *bs'-def*:
*set* (*as'*@*U*@*bs'*@*V*@*cs'*) = *xs distinct* (*as'*@*U*@*bs'*@*V*@*cs'*) *take 1* (*as'*@*U*@*bs'*@*V*@*cs'*) = [*r*]
$\forall$ *as bs cs*. *P* (*as* @ *U* @ *bs* @ *V* @ *cs*) $\longrightarrow$
*cost* (*rev* (*as'* @ *U* @ *bs'* @ *V* @ *cs'*)) $\leq$ *cost* (*rev* (*as* @ *U* @ *bs* @ *V* @ *cs*))
**using** *valid-UV-lists-argmin-r-ex-noP'*[*OF assms*(3−8)] *assms*(9,10) **unfolding** *P-def* **by** *blast*
**then consider** *U* = [] | *V* = [] $\vee$ *bs'* = []
| *rank* (*rev bs'*) $\leq$ *rank* (*rev U*) *U* $\neq$ [] *bs'* $\neq$ []
| *rank* (*rev U*) $\leq$ *rank* (*rev bs'*) *U* $\neq$ [] *V* $\neq$ [] *bs'* $\neq$ []
**by** *fastforce*
**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then have** $\forall$ *as bs cs*. *P* (*as* @ *U* @ *bs* @ *V* @ *cs*) $\longrightarrow$
*cost* (*rev* ((*as'*@*bs'*)@*U*@*V*@*cs'*)) $\leq$ *cost* (*rev* (*as* @ *U* @ *bs* @ *V* @ *cs*))
**using** *bs'-def*(*4*) **by** *simp*
**moreover have** *set* ((*as'*@*bs'*)@*U*@*V*@*cs'*) = *xs* **using** *bs'-def*(*1*) **by** *auto*
**moreover have** *distinct* ((*as'*@*bs'*)@*U*@*V*@*cs'*) **using** *bs'-def*(*2*) **by** *auto*
**moreover have** *take 1* ((*as'*@*bs'*)@*U*@*V*@*cs'*) = [*r*] **using** *bs'-def*(*3*) *1* **by** *auto*
**ultimately show** *?thesis* **unfolding** *P-def* **by** *blast*
**next**
**case** *2*
**then have** $\forall$ *as bs cs*. *P* (*as* @ *U* @ *bs* @ *V* @ *cs*) $\longrightarrow$
*cost* (*rev* (*as'*@*U*@*V*@*bs'*@*cs'*)) $\leq$ *cost* (*rev* (*as* @ *U* @ *bs* @ *V* @ *cs*))
**using** *bs'-def*(*4*) **by** *auto*
**moreover have** *set* (*as'*@*U*@*V*@*bs'*@*cs'*) = *xs* **using** *bs'-def*(*1*) **by** *auto*
**moreover have** *distinct* (*as'*@*U*@*V*@*bs'*@*cs'*) **using** *bs'-def*(*2*) **by** *auto*
**moreover have** *take 1* (*as'*@*U*@*V*@*bs'*@*cs'*) = [*r*] **using** *bs'-def*(*3*) *2* **by** *auto*
**ultimately show** *?thesis* **unfolding** *P-def* **by** *blast*
**next**
**case** *3*
**have** *0*: *distinct* (*as'*@*bs'*@*U*@*V*@*cs'*) **using** *bs'-def*(*2*) **by** *auto*
**have** *1*: *take 1* (*as'*@*bs'*@*U*@*V*@*cs'*) = [*r*]
**using** *bs'-def*(*3*) *assms*(*9*) *3*(*2*) *take1-split-nelem-nempty*[*of as' U bs'*@*V*@*cs'*] **by** *simp*
**then have** *cost* (*rev* (*as'*@*bs'*@*U*@*V*@*cs'*)) $\leq$ *cost* (*rev* (*as'*@*U*@*bs'*@*V*@*cs'*))
**using** *asi-le-rfst*[*OF assms*(*1*) *3*(*1*,*3*,*2*) *0*] *bs'-def*(*3*) **by** *blast*
**then have** $\forall$ *as bs cs*. *P* (*as* @ *U* @ *bs* @ *V* @ *cs*) $\longrightarrow$
*cost* (*rev* ((*as'*@*bs'*)@*U*@*V*@*cs'*)) $\leq$ *cost* (*rev* (*as* @ *U* @ *bs* @ *V* @ *cs*))
**using** *bs'-def*(*4*) **by** *fastforce*

```
    moreover have set ((as'@bs')@U@V@cs') = xs using bs'-def(1) by auto
    moreover have distinct ((as'@bs')@U@V@cs') using 0 by simp
    moreover have take 1 ((as'@bs')@U@V@cs') = [r] using 1 by simp
    ultimately show ?thesis using P-def by blast
  next
    case 4
    then have 3: rank (rev V) ≤ rank (rev bs') using assms(2) by simp
    have 0: distinct ((as'@U)@V@bs'@cs') using bs'-def(2) by auto
    have 1: take 1 (as'@U@V@bs'@cs') = [r]
     using bs'-def(3) assms(9) 4(2) take1-split-nelem-nempty[of as' U bs'@V@cs']
by simp
   then have cost (rev (as'@U@V@bs'@cs')) ≤ cost (rev ((as'@U)@bs'@V@cs'))
      using asi-le-rfst[OF assms(1) 3 4(3,4) 0] bs'-def(3) by simp
    then have ∀ as bs cs. P (as @ U @ bs @ V @ cs) ⟶
        cost (rev (as'@U@V@bs'@cs')) ≤ cost (rev (as @ U @ bs @ V @ cs))
      using bs'-def(4) by fastforce
    moreover have set (as'@U@V@bs'@cs') = xs using bs'-def(1) by auto
    moreover have distinct (as'@U@V@bs'@cs') using 0 by simp
    ultimately show ?thesis using P-def 1 by blast
  qed
qed

fun combine-lists-P :: ('a list ⇒ bool) ⇒ 'a list ⇒ 'a list list ⇒ 'a list list where
  combine-lists-P - y [] = [y]
| combine-lists-P P y (x#xs) = (if P (x@y) then combine-lists-P P (x@y) xs else
(x@y)#xs)

fun make-list-P :: ('a list ⇒ bool) ⇒ 'a list list ⇒ 'a list list ⇒ 'a list list where
  make-list-P P acc xs = (case List.extract P xs of
      None ⇒ rev acc @ xs
    | Some (as,y,bs) ⇒ make-list-P P (combine-lists-P P y (rev as @ acc)) bs)

lemma combine-lists-concat-rev-eq: concat (rev (combine-lists-P P y xs)) = concat
(rev xs) @ y
  by (induction P y xs rule: combine-lists-P.induct) auto

lemma make-list-concat-rev-eq: concat (make-list-P P acc xs) = concat (rev acc)
@ concat xs
proof(induction P acc xs rule: make-list-P.induct)
  case (1 P acc xs)
  then show ?case
  proof(cases List.extract P xs)
    case (Some a)
    then obtain as x bs where x-def[simp]: a = (as,x,bs) by(cases a) auto
    then have concat (make-list-P P acc xs)
      = concat (rev (combine-lists-P P x (rev as @ acc))) @ concat bs
      using 1 Some by simp
    also have ... = concat (rev acc) @ concat (as@x#bs)
      using combine-lists-concat-rev-eq[of P] by simp
```

**finally show** *?thesis* **using** *Some extract-SomeE* **by** *force*
**qed**(*simp*)
**qed**

**lemma** *combine-lists-sublists*:
$\exists x \in \{y\} \cup$ *set xs. sublist as x* $\Longrightarrow \exists x \in$ *set* (*combine-lists-P P y xs*). *sublist as x*
**proof** (*induction P y xs rule*: *combine-lists-P.induct*)
**case** (*2 P y x xs*)
**then show** *?case*
**proof**(*cases sublist as x* $\vee$ *sublist as y*)
**case** *True*
**then have** *sublist as* (*x*@*y*) **using** *sublist-order.dual-order.trans* **by** *blast*
**then show** *?thesis* **using** *2* **by** *force*
**next**
**case** *False*
**then show** *?thesis* **using** *2* **by** *simp*
**qed**
**qed**(*simp*)

**lemma** *make-list-sublists*:
$\exists x \in$ *set acc* $\cup$ *set xs. sublist cs x* $\Longrightarrow \exists x \in$ *set* (*make-list-P P acc xs*). *sublist cs x*
**proof**(*induction P acc xs rule*: *make-list-P.induct*)
**case** (*1 P acc xs*)
**then show** *?case*
**proof**(*cases List.extract P xs*)
**case** (*Some a*)
**then obtain** *as x bs* **where** *x-def*[*simp*]: *a* = (*as*,*x*,*bs*) **by**(*cases a*) *auto*
**then have** *make-list-P P acc xs* = *make-list-P P* (*combine-lists-P P x* (*rev as* @ *acc*)) *bs*
**using** *Some* **by** *simp*
**then have** $\exists a \in$ *set* (*combine-lists-P P x* (*rev as* @ *acc*)) $\cup$ *set bs. sublist cs a*
**using** *Some combine-lists-sublists*[*of x rev as* @ *acc cs*] *1.prems*
**by** (*auto simp*: *extract-Some-iff*)
**then show** *?thesis* **using** *1 Some* **by** *simp*
**qed**(*simp*)
**qed**

**lemma** *combine-lists-nempty*: $[\![[] \notin$ *set xs*; *y* $\neq$ []$]\!] \Longrightarrow$ [] $\notin$ *set* (*combine-lists-P P y xs*)
**by** (*induction P y xs rule*: *combine-lists-P.induct*) *auto*

**lemma** *make-list-nempty*:
$[\![[] \notin$ *set acc*; [] $\notin$ *set xs*$]\!] \Longrightarrow$ [] $\notin$ *set* (*make-list-P P acc xs*)
**proof** (*induction P acc xs rule*: *make-list-P.induct*)
**case** (*1 P acc xs*)
**show** *?case*
**proof**(*cases List.extract P xs*)

```
    case None
    then show ?thesis using 1 by simp
  next
    case (Some a)
   then show ?thesis using 1 by (auto simp: extract-Some-iff combine-lists-nempty)
  qed
qed

lemma combine-lists-notP:
 ∀ x∈set xs. ¬P x ⟹ (∃ x. combine-lists-P P y xs = [x]) ∨ (∀ x∈set (combine-lists-P
P y xs). ¬P x)
  by (induction P y xs rule: combine-lists-P.induct) auto

lemma combine-lists-single: xs = [x] ⟹ combine-lists-P P y xs = [x@y]
  by auto

lemma combine-lists-lastP:
  P (last xs) ⟹ (∃ x. combine-lists-P P y xs = [x]) ∨ (P (last (combine-lists-P P
y xs)))
  by (induction P y xs rule: combine-lists-P.induct) auto

lemma make-list-notP:
  ⟦(∀ x ∈ set acc. ¬P x) ∨ P (last acc)⟧
    ⟹ (∀ x∈set (make-list-P P acc xs). ¬P x) ∨ (∃ y ys. make-list-P P acc xs = y
# ys ∧ P y)
proof(induction P acc xs rule: make-list-P.induct)
  case (1 P acc xs)
  then show ?case
  proof(cases List.extract P xs)
    case None
    then show ?thesis
    proof(cases ∀ x ∈ set acc. ¬P x)
      case True
      from None have ∀ x ∈ set xs. ¬ P x by (simp add: extract-None-iff)
      then show ?thesis using True 1.prems None by auto
    next
      case False
      then have acc ≠ [] by auto
       then have make-list-P P acc xs = last acc # rev (butlast acc) @ xs using
None by simp
      then show ?thesis using False 1.prems by blast
    qed
  next
    case (Some a)
    then obtain as x bs where x-def[simp]: a = (as,x,bs) by(cases a) auto
    show ?thesis
    proof(cases ∀ x ∈ set acc. ¬P x)
      case True
        then have ∀ x ∈ set (rev as @ acc). ¬P x using Some by (auto simp:
```

*extract-Some-iff*)
**then have** ($\forall$ *x*$\in$*set* (*combine-lists-P P x* (*rev as* @ *acc*)). $\neg$ *P x*)
$\vee$ *P* (*last* (*combine-lists-P P x* (*rev as* @ *acc*)))
**using** *combine-lists-notP*[*of rev as* @ *acc P*] **by** *force*
**then show** *?thesis* **using** *1.IH Some* **by** *simp*
**next**
**case** *False*
**then have** *P* (*last acc*) $\wedge$ *acc* $\neq$ [] **using** *1.prems* **by** *auto*
**then have** *P* (*last* (*rev as* @ *acc*)) **using** *1.prems* **by** *simp*
**then have** ($\forall$ *x*$\in$*set* (*combine-lists-P P x* (*rev as* @ *acc*)). $\neg$ *P x*)
$\vee$ *P* (*last* (*combine-lists-P P x* (*rev as* @ *acc*)))
**using** *combine-lists-lastP*[*of P*] **by** *force*
**then show** *?thesis* **using** *1.IH Some* **by** *simp*
**qed**
**qed**
**qed**

**corollary** *make-list-notP-empty-acc*:
($\forall$ *x*$\in$*set* (*make-list-P P* [] *xs*). $\neg$*P x*) $\vee$ ($\exists$ *y ys*. *make-list-P P* [] *xs* = *y* # *ys* $\wedge$ *P y*)
**using** *make-list-notP*[*of* []] **by** *auto*

**definition** *unique-set-r* :: *'a* $\Rightarrow$ *'a list set* $\Rightarrow$ *'a list* $\Rightarrow$ *bool* **where**
*unique-set-r r Y ys* $\longleftrightarrow$ *set ys* = $\bigcup$(*set* ' *Y*) $\wedge$ *distinct ys* $\wedge$ *take 1 ys* = [*r*]

**context** *directed-tree*
**begin**

**definition** *fwd-sub* :: *'a* $\Rightarrow$ *'a list set* $\Rightarrow$ *'a list* $\Rightarrow$ *bool* **where**
*fwd-sub r Y ys* $\longleftrightarrow$ *unique-set-r r Y ys* $\wedge$ *forward ys* $\wedge$ ($\forall$ *xs* $\in$ *Y*. *sublist xs ys*)

**lemma** *distinct-mid-unique1*: ⟦*distinct* (*xs*@*U*@*ys*); *U*$\neq$[]; *xs*@*U*@*ys* = *as*@*U*@*bs*⟧
$\Longrightarrow$ *as* = *xs*
**using** *distinct-app-trans-r distinct-ys-not-xs*[*of xs U*@*ys*] *hd-append2*[*of U*] *append-is-Nil-conv*[*of U*]
**by** (*metis append-Cons-eq-iff distinct.simps*(*2*) *list.exhaust-sel list.set-sel*(*1*))

**lemma** *distinct-mid-unique2*: ⟦*distinct* (*xs*@*U*@*ys*); *U*$\neq$[]; *xs*@*U*@*ys* = *as*@*U*@*bs*⟧
$\Longrightarrow$ *ys* = *bs*
**using** *distinct-mid-unique1* **by** *blast*

**lemma** *concat-all-sublist*: $\forall$ *x* $\in$ *set xs*. *sublist x* (*concat xs*)
**using** *split-list* **by** *force*

**lemma** *concat-all-sublist-rev*: $\forall$ *x* $\in$ *set xs*. *sublist x* (*concat* (*rev xs*))
**using** *split-list* **by** *force*

**lemma** *concat-all-sublist1*:
**assumes** *distinct* (*as*@*U*@*bs*)

**and** *concat cs @ U @ concat ds = as@U@bs*
**and** $U \neq []$
**and** *set (cs@U#ds) = Y*
**shows** $\exists X.\ X \subseteq Y \wedge set\ as = \bigcup (set\ `\ X) \wedge (\forall xs \in X.\ sublist\ xs\ as)$
**proof** −
**have** *eq*: *concat cs = as*
**using** *distinct-mid-unique1*[*of concat cs U concat ds*] *assms*(*1*−*3*) **by** *simp*
**then have** $\forall xs \in set\ cs.\ sublist\ xs\ as$ **using** *concat-all-sublist* **by** *blast*
**then show** *?thesis* **using** *eq assms*(*4*) **by** *fastforce*
**qed**

**lemma** *concat-all-sublist2*:
**assumes** *distinct (as@U@bs)*
**and** *concat cs @ U @ concat ds = as@U@bs*
**and** $U \neq []$
**and** *set (cs@U#ds) = Y*
**shows** $\exists X.\ X \subseteq Y \wedge set\ bs = \bigcup (set\ `\ X) \wedge (\forall xs \in X.\ sublist\ xs\ bs)$
**proof** −
**have** *eq*: *concat ds = bs*
**using** *distinct-mid-unique1*[*of concat cs U concat ds*] *assms*(*1*−*3*) **by** *simp*
**then have** $\forall xs \in set\ ds.\ sublist\ xs\ bs$ **using** *concat-all-sublist* **by** *blast*
**then show** *?thesis* **using** *eq assms*(*4*) **by** *fastforce*
**qed**

**lemma** *concat-split-mid*:
**assumes** $\forall xs \in Y.\ \forall ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
**and** *finite Y*
**and** $U \in Y$
**and** *distinct (as@U@bs)*
**and** *set (as@U@bs) =* $\bigcup (set\ `\ Y)$
**and** $\forall xs \in Y.\ sublist\ xs\ (as@U@bs)$
**and** $U \neq []$
**shows** $\exists cs\ ds.\ concat\ cs = as \wedge concat\ ds = bs \wedge set\ (cs@U\#ds) = Y \wedge$
*distinct (cs@U#ds)*
**proof** −
**obtain** *ys* **where** *ys-def*: *set ys = Y concat ys = as@U@bs distinct ys*
**using** *list-of-sublist-concat-eq*[*OF assms*(*1*,*6*,*4*,*5*,*2*)] **by** *blast*
**then obtain** *cs ds* **where** *cs-def*: *cs@U#ds = ys*
**using** *assms*(*3*) *in-set-conv-decomp-first*[*of U ys*] **by** *blast*
**then have** *List.extract* ((=) *U*) *ys = Some* (*cs*,*U*,*ds*)
**using** *extract-Some-iff*[*of* (=) *U*] *ys-def*(*3*) **by** *auto*
**then have** *concat cs @ U @ concat ds = as@U@bs* **using** *ys-def*(*2*) *cs-def* **by** *auto*
**then have** *concat cs = as* ∧ *concat ds = bs*
**using** *distinct-mid-unique1*[*of concat cs U*] *assms*(*4*,*7*) **by** *auto*
**then show** *?thesis* **using** *ys-def*(*1*,*3*) *cs-def* **by** *blast*
**qed**

**lemma** *mid-all-sublists-set1*:

**assumes** $\forall\, xs \in Y.\ \forall\, ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
**and** *finite* $Y$
**and** $U \in Y$
**and** *distinct* $(as@U@bs)$
**and** $set\ (as@U@bs) = \bigcup (set\ `\ Y)$
**and** $\forall\, xs \in Y.\ sublist\ xs\ (as@U@bs)$
**and** $U \neq []$
**shows** $\exists\, X.\ X \subseteq Y \wedge set\ as = \bigcup (set\ `\ X) \wedge (\forall\, xs \in X.\ sublist\ xs\ as)$
**proof** −
**obtain** *ys* **where** *ys-def*: $set\ ys = Y$ *concat* $ys = as@U@bs$ *distinct* $ys$
**using** *list-of-sublist-concat-eq*[*OF assms*(1,6,4,5,2)] **by** *blast*
**then obtain** *cs ds* **where** *cs-def*: $cs@U\#ds = ys$
**using** *assms*(3) *in-set-conv-decomp-first*[*of U ys*] **by** *blast*
**then have** *List.extract* $((=)\ U)\ ys = Some\ (cs,U,ds)$
**using** *extract-Some-iff*[*of* $(=)\ U$] *ys-def*(3) **by** *auto*
**then have** *concat cs* @ *U* @ *concat ds* = $as@U@bs$ **using** *ys-def*(2) *cs-def* **by** *auto*
**then show** *?thesis* **using** *cs-def ys-def*(1) *concat-all-sublist1*[*OF assms*(4)] *assms*(7) **by** *force*
**qed**

**lemma** *mid-all-sublists-set2*:
**assumes** $\forall\, xs \in Y.\ \forall\, ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
**and** *finite* $Y$
**and** $U \in Y$
**and** *distinct* $(as@U@bs)$
**and** $set\ (as@U@bs) = \bigcup (set\ `\ Y)$
**and** $\forall\, xs \in Y.\ sublist\ xs\ (as@U@bs)$
**and** $U \neq []$
**shows** $\exists\, X.\ X \subseteq Y \wedge set\ bs = \bigcup (set\ `\ X) \wedge (\forall\, xs \in X.\ sublist\ xs\ bs)$
**proof** −
**obtain** *ys* **where** *ys-def*: $set\ ys = Y$ *concat* $ys = as@U@bs$ *distinct* $ys$
**using** *list-of-sublist-concat-eq*[*OF assms*(1,6,4,5,2)] **by** *blast*
**then obtain** *cs ds* **where** *cs-def*: $cs@U\#ds = ys$
**using** *assms*(3) *in-set-conv-decomp-first*[*of U ys*] **by** *blast*
**then have** *List.extract* $((=)\ U)\ ys = Some\ (cs,U,ds)$
**using** *extract-Some-iff*[*of* $(=)\ U$] *ys-def*(3) **by** *auto*
**then have** *concat cs* @ *U* @ *concat ds* = $as@U@bs$ **using** *ys-def*(2) *cs-def* **by** *auto*
**then show** *?thesis* **using** *cs-def ys-def*(1) *concat-all-sublist2*[*OF assms*(4)] *assms*(7) **by** *force*
**qed**

**lemma** *nonempty-notin-distinct-prefix*:
**assumes** *distinct* $(as@bs@V@cs)$ **and** *concat* $as' = as$ **and** $V \neq []$
**shows** $V \notin set\ as'$
**proof**
**assume** $V \in set\ as'$
**then have** $set\ V \subseteq set\ as$ **using** *assms*(2) **by** *auto*

**then have** *set as* ∩ *set V* ≠ {} **using** *assms*(*3*) **by** (*simp add*: *Int-absorb1*)
**then show** *False* **using** *assms*(*1*) **by** *auto*
**qed**

**lemma** *concat-split-UV*:
**assumes** ∀ *xs* ∈ *Y*. ∀ *ys* ∈ *Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}
**and** *finite Y*
**and** *U* ∈ *Y*
**and** *V* ∈ *Y*
**and** *distinct* (*as*@*U*@*bs*@*V*@*cs*)
**and** *set* (*as*@*U*@*bs*@*V*@*cs*) = ⋃(*set* ‘ *Y*)
**and** ∀ *xs* ∈ *Y*. *sublist xs* (*as*@*U*@*bs*@*V*@*cs*)
**and** *U* ≠ []
**and** *V* ≠ []
**shows** ∃ *as*′ *bs*′ *cs*′. *concat as*′ = *as* ∧ *concat bs*′ = *bs* ∧ *concat cs*′ = *cs*
∧ *set* (*as*′@*U*#*bs*′@*V*#*cs*′) = *Y* ∧ *distinct* (*as*′@*U*#*bs*′@*V*#*cs*′)
**proof** −
**obtain** *as*′ *ds* **where** *as*′-*def*:
*concat as*′ = *as concat ds* = *bs*@*V*@*cs set* (*as*′@*U*#*ds*) = *Y distinct* (*as*′@*U*#*ds*)
**using** *concat-split-mid*[*OF assms*(*1*−*3*,*5*−*8*)] **by** *auto*
**have** *0*: *distinct* (*bs*@*V*@*cs*) **using** *assms*(*5*) **by** *simp*
**have** *V* ∉ *set as*′
**using** *assms*(*5*,*9*) *as*′-*def*(*1*) *nonempty-notin-distinct-prefix*[*of as U*@*bs*] **by** *auto*
**moreover have** *V* ≠ *U* **using** *assms*(*5*,*8*,*9*) *empty-if-sublist-dsjnt*[*of U*] **by** *auto*
**ultimately have** *V* ∈ *set ds* **using** *as*′-*def*(*3*) *assms*(*4*) **by** *auto*
**then show** *?thesis*
**using** *as*′-*def 0 assms*(*9*) *concat-append distinct-mid-unique1*
**by** (*metis concat.simps*(*2*) *distinct-mid-unique2 split-list*)
**qed**

**lemma** *cost-decr-if-noarc-lessrank*:
**assumes** *asi rank r cost*
**and** *b* ≠ []
**and** *r* ∉ *set U*
**and** *U* ≠ []
**and** *set* (*as*@*U*@*bs*@*cs*) = ⋃(*set* ‘ *Y*)
**and** *distinct* (*as*@*U*@*bs*@*cs*)
**and** *take 1* (*as*@*U*@*bs*@*cs*) = [*r*]
**and** *forward* (*as*@*U*@*bs*@*cs*)
**and** *concat* (*b*#*bs*′) = *bs*
**and** (∀ *xs* ∈ *Y*. *sublist xs as* ∨ *sublist xs U*
∨ (∃ *x* ∈ *set* (*b*#*bs*′). *sublist xs x*) ∨ *sublist xs cs*)
**and** ¬(∃ *x* ∈ *set U*. ∃ *y* ∈ *set b*. *x* →$_T$ *y*)
**and** *rank* (*rev b*) < *rank* (*rev U*)
**shows** *fwd-sub r Y* (*as*@*b*@*U*@*concat bs*′@*cs*)
∧ *cost* (*rev* (*as*@*b*@*U*@*concat bs*′@*cs*)) < *cost* (*rev* (*as*@*U*@*bs*@*cs*))
**proof** −
**have** *rank-yU*: *rank* (*rev b*) < *rank* (*rev U*) **using** *assms*(*12*) **by** *simp*

**have** *0*: *take 1* (*as*@*b*@*U*@*concat bs'*@*cs*) = [*r*]
**using** *take1-singleton-app take1-split-nelem-nempty*[*OF assms*(*7*,*4*,*3*)] **by** *fast*
**have** *1*: *distinct* (*as*@*b*@*U*@ *concat bs'*@*cs*) **using** *assms*(*6*,*9*) **by** *force*
**have** *take 1* (*as*@*U*@*b*@*concat bs'*@*cs*) = [*r*] **using** *assms*(*7*,*9*) **by** *force*
**then have** *cost-lt*: *cost* (*rev* (*as*@*b*@*U*@*concat bs'*@*cs*)) < *cost* (*rev* (*as*@*U*@*bs*@*cs*))
**using** *asi-lt-rfst*[*OF assms*(*1*) *rank-yU assms*(*2*,*4*) *1 0*] *assms*(*9*) **by** *fastforce*
**have** *P*: *set* (*as*@*b*@*U*@*concat bs'*@*cs*) = ⋃(*set* ‘ *Y*) **using** *assms*(*5*,*9*) **by** *fastforce*
**then have** *P*: *unique-set-r r Y* (*as*@*b*@*U*@*concat bs'*@*cs*)
**using** *0 1* **unfolding** *unique-set-r-def* **by** *blast*
**have** (∀ *xs* ∈ *Y*. *sublist xs as* ∨ *sublist xs U* ∨ *sublist xs b*
∨ *sublist xs* (*concat bs'*) ∨ *sublist xs cs*)
**using** *assms*(*10*) *concat-all-sublist*[*of bs'*]
*sublist-order.dual-order.trans*[**where** *a* = *concat bs'*] **by** *auto*
**then have** *all-sub*: ∀ *xs* ∈ *Y*. *sublist xs* (*as*@*b*@*U*@*concat bs'*@*cs*)
**by** (*metis sublist-order.order.trans sublist-append-leftI sublist-append-rightI*)
**have** *as* ≠ [] **using** *take1-split-nelem-nempty*[*OF assms*(*7*,*4*,*3*)] **by** *force*
**then have** *forward* (*as*@*b*@*U*@*concat bs'*@*cs*)
**using** *move-mid-forward-if-noarc assms*(*8*,*9*,*11*) **by** *auto*
**then show** *?thesis* **using** *assms*(*12*) *P all-sub cost-lt fwd-sub-def* **by** *blast*
**qed**

**lemma** *cost-decr-if-noarc-lessrank'*:
**assumes** *asi rank r cost*
**and** *b* ≠ []
**and** *r* ∉ *set U*
**and** *U* ≠ []
**and** *set* (*as*@*U*@*bs*@*cs*) = ⋃(*set* ‘ *Y*)
**and** *distinct* (*as*@*U*@*bs*@*cs*)
**and** *take 1* (*as*@*U*@*bs*@*cs*) = [*r*]
**and** *forward* (*as*@*U*@*bs*@*cs*)
**and** *concat* (*b*#*bs'*) = *bs*
**and** (∀ *xs* ∈ *Y*. *sublist xs as* ∨ *sublist xs U*
∨ (∃ *x* ∈ *set* (*b*#*bs'*). *sublist xs x*) ∨ *sublist xs cs*)
**and** ¬(∃ *x* ∈ *set U*. ∃ *y* ∈ *set b*. *x* →$_T$ *y*)
**and** *rank* (*rev b*) < *rank* (*rev V*)
**and** *rank* (*rev V*) ≤ *rank* (*rev U*)
**shows** *fwd-sub r Y* (*as*@*b*@*U*@*concat bs'*@*cs*)
∧ *cost* (*rev* (*as*@*b*@*U*@*concat bs'*@*cs*)) < *cost* (*rev* (*as*@*U*@*bs*@*cs*))
**using** *cost-decr-if-noarc-lessrank*[*OF assms*(*1*−*11*)] *assms*(*12*,*13*) **by** *simp*

**lemma** *sublist-exists-append*:
∃ *a*∈*set* ((*x* # *xs*) @ [*b*]). *sublist ys a* ⟹ ∃ *a*∈*set*(*xs* @ [*x*@*b*]). *sublist ys a*
**using** *sublist-order.dual-order.trans* **by** *auto*

**lemma** *sublist-set-concat-cases*:
∃ *a*∈*set* ((*x* # *xs*) @ [*b*]). *sublist ys a* ⟹ *sublist ys* (*concat* (*rev xs*)) ∨ *sublist ys x* ∨ *sublist ys b*
**using** *sublist-order.dual-order.trans concat-all-sublist-rev*[*of xs*] **by** *auto*

**lemma** *sublist-set-concat-or-cases-aux1*:
*sublist ys as* $\vee$ *sublist ys U* $\vee$ *sublist ys cs*
$\Longrightarrow$ *sublist ys* (*as* @ *U* @ *concat* (*rev xs*)) $\vee$ *sublist ys cs*
**using** *sublist-order.dual-order.trans* **by** *blast*

**lemma** *sublist-set-concat-or-cases-aux2*:
$\exists\, a{\in}set\ ((x\ \#\ xs)\ @\ [b]).$ *sublist ys a*
$\Longrightarrow$ *sublist ys* (*as* @ *U* @ *concat* (*rev xs*)) $\vee$ *sublist ys x* $\vee$ *sublist ys b*
**using** *sublist-set-concat-cases*[*of x xs b ys*] *sublist-order.dual-order.trans* **by** *blast*

**lemma** *sublist-set-concat-or-cases*:
*sublist ys as* $\vee$ *sublist ys U* $\vee$ ($\exists\, a{\in}set\ ((x\#xs)\ @\ [b]).$ *sublist ys a*) $\vee$ *sublist ys cs* $\Longrightarrow$
*sublist ys* (*as*@*U*@ *concat* (*rev xs*)) $\vee$ *sublist ys x* $\vee$ ($\exists\, a{\in}set\ [b].$ *sublist ys a*) $\vee$ *sublist ys cs*
**using** *sublist-set-concat-or-cases-aux1*[*of ys as U cs*] *sublist-set-concat-or-cases-aux2*[*of x xs b ys*]
**by** *auto*

**corollary** *not-reachable1-append-if-not-old*:
⟦$\neg$ ($\exists\, z{\in}set\ U.\ \exists\, y{\in}set\ b.\ z\ \rightarrow^+{}_T\ y$); *set U* $\cap$ *set x* = {}; *forward x*;
$\exists\, z{\in}set\ x.\ \exists\, y{\in}set\ b.\ z\ \rightarrow_T\ y$⟧
$\Longrightarrow$ $\neg$ ($\exists\, z{\in}set\ U.\ \exists\, y{\in}set\ (x@b).\ z\ \rightarrow^+{}_T\ y$)
**using** *reachable1-append-old-if-arcU*[*of x b U*] **by** *auto*

**lemma** *combine-lists-notP*:
**assumes** *asi rank r cost*
**and** $b \neq []$
**and** $r \notin$ *set U*
**and** $U \neq []$
**and** *set* (*as*@*U*@*bs*@*cs*) = $\bigcup$(*set* ‘ *Y*)
**and** *distinct* (*as*@*U*@*bs*@*cs*)
**and** *take 1* (*as*@*U*@*bs*@*cs*) = [*r*]
**and** *forward* (*as*@*U*@*bs*@*cs*)
**and** *concat* (*rev ys* @ [*b*]) = *bs*
**and** ($\forall$ *xs* $\in$ *Y*. *sublist xs as* $\vee$ *sublist xs U*
$\vee$ ($\exists\, x \in$ *set* (*ys* @ [*b*]). *sublist xs x*) $\vee$ *sublist xs cs*)
**and** *rank* (*rev V*) $\leq$ *rank* (*rev U*)
**and** $\neg(\exists\, x \in$ *set U*. $\exists\, y \in$ *set b*. *x* $\rightarrow^+{}_T$ *y*)
**and** *rank* (*rev b*) < *rank* (*rev V*)
**and** *P* = ($\lambda$*x*. *rank* (*rev x*) < *rank* (*rev V*))
**and** $\forall\, x{\in}set\ ys.\ \neg P\ x$
**and** $\forall$ *xs*. *fwd-sub r Y xs* $\longrightarrow$ *cost* (*rev* (*as*@*U*@*bs*@*cs*)) $\leq$ *cost* (*rev xs*)
**and** $\forall\, x \in$ *set ys*. $x \neq []$
**and** $\forall\, x \in$ *set ys*. *forward x*
**and** *forward b*
**shows** $\forall\, x{\in}set$ (*combine-lists-P P b ys*). $\neg P\ x \wedge$ *forward x*
**using** *assms* **proof**(*induction P b ys rule*: *combine-lists-P.induct*)

**case** (*1 P b*)
**have** *0*: *concat* (*b*#[]) = *bs* **using** *1.prems*(*9*) **by** *simp*
**have** *2*: (∀ *xs* ∈ *Y*. *sublist xs as* ∨ *sublist xs U*
∨ (∃ *x* ∈ *set* ([*b*]). *sublist xs x*) ∨ *sublist xs cs*) **using** *1.prems*(*10*) **by** *simp*
**have** *3*: ¬ (∃ *x*∈*set U*. ∃ *y*∈*set b*. *x* →$_T$ *y*) **using** *1.prems*(*12*) **by** *blast*
**show** *?case*
**using** *cost-decr-if-noarc-lessrank'*[*OF 1*(*1*−*8*) *0 2 3 1*(*13*,*11*)] *1*(*16*) **by** *auto*
**next**
**case** (*2 P b x xs*)
**have** *take 1 as* = [*r*] **using** *2.prems*(*3*,*4*,*7*) *take1-split-nelem-empty* **by** *fast*
**then have** *r* ∈ *set as* **using** *in-set-takeD*[*of r 1*] **by** *simp*
**then have** *r* ∉ *set x* **using** *2.prems*(*6*,*9*) **by** *force*
**then have** *x* ≠ [] **using** *2.prems*(*17*) **by** *simp*

Arc between x and b otherwise not optimal.

**have** *4*: *as*@*U*@*bs*@*cs* = (*as*@*U*@*concat* (*rev xs*)) @ *x* @ *b* @ *cs* **using** *2.prems*(*9*)
**by** *simp*
**have** *set*: *set* ((*as*@*U*@*concat* (*rev xs*)) @ *x* @ *b* @ *cs*) = ⋃ (*set* ‘ *Y*)
**using** *2.prems*(*5*) *4* **by** *simp*
**have** *dst*: *distinct* ((*as*@*U*@*concat* (*rev xs*)) @ *x* @ *b* @ *cs*) **using** *2.prems*(*6*) *4*
**by** *simp*
**have** *tk1*: *take 1* ((*as*@*U*@*concat* (*rev xs*)) @ *x* @ *b* @ *cs*) = [*r*] **using** *2.prems*(*7*)
*4* **by** *simp*
**have** *fwd*: *forward* ((*as*@*U*@*concat* (*rev xs*)) @ *x* @ *b* @ *cs*) **using** *2.prems*(*8*) *4*
**by** *simp*
**have** *cnct*: *concat* (*b* # []) = *b* **by** *simp*
**have** *sblst*: ∀ *xs'* ∈ *Y*. *sublist xs'* (*as* @ *U* @ *concat* (*rev xs*)) ∨ *sublist xs' x*
∨ (∃ *a*∈*set* [*b*]. *sublist xs' a*) ∨ *sublist xs' cs*
**using** *2.prems*(*10*) *sublist-set-concat-or-cases*[**where** *as* = *as*] **by** *simp*
**have** *rank* (*rev b*) < *rank* (*rev x*) **using** *2.prems*(*13*−*15*) **by** *simp*
**then have** *arc-xb*: ∃ *z*∈*set x*. ∃ *y*∈*set b*. *z* →$_T$ *y*
**using** *2.prems*(*16*) *4*
*cost-decr-if-noarc-lessrank*[*OF 2*(*2*,*3*) ‹*r*∉*set x*› ‹*x*≠[]› *set dst tk1 fwd cnct*
*sblst*]
**by** *fastforce*
**have** *set x* ∩ *set b* = {} **using** *dst* **by** *auto*
**then have** *fwd*: *forward* (*x*@*b*) **using** *forward-app' arc-xb 2.prems*(*18*,*19*) **by**
*simp*
**show** *?case*
**proof**(*cases P* (*x* @ *b*))
**case** *True*
**have** *0*: *x* @ *b* ≠ [] **using** *2.prems*(*2*) **by** *blast*
**have** *1*: *concat* (*rev xs* @ [*x* @ *b*]) = *bs* **using** *2.prems*(*9*) **by** *simp*
**have** *3*: ∀ *xs'* ∈ *Y*. *sublist xs' as* ∨ *sublist xs' U*
∨ (∃ *a*∈*set* (*xs* @ [*x* @ *b*]). *sublist xs' a*) ∨ *sublist xs' cs*
**using** *2.prems*(*10*) *sublist-exists-append* **by** *fast*
**have** *set U* ∩ *set x* = {} **using** *4 2.prems*(*6*) **by** *force*
**then have** *4*: ¬ (∃ *z*∈*set U*. ∃ *y*∈*set* (*x* @ *b*). *z* →$^+_T$ *y*)
**using** *not-reachable1-append-if-not-old*[*OF 2.prems*(*12*)] *2.prems*(*18*) *arc-xb*

**by** *simp*
  **have** *5*: *rank* (*rev* (*x* @ *b*)) < *rank* (*rev V*) **using** *True 2.prems*(*14*) **by** *simp*
  **show** *?thesis*
   **using** *2.IH*[*OF True 2*(*2*) *0 2*(*4*−*9*) *1 3 2*(*12*) *4 5 2*(*15*)] *2*(*16*−*19*) *fwd* **by** *auto*
 **next**
  **case** *False*
  **then show** *?thesis* **using** *2.prems*(*15*,*18*) *fwd* **by** *simp*
 **qed**
**qed**

**lemma** *sublist-app-l*: *sublist ys cs* ⟹ *sublist ys* (*xs* @ *cs*)
 **using** *sublist-order.dual-order.trans* **by** *blast*

**lemma** *sublist-split-concat*:
 **assumes** *a* ∈ *set* (*acc* @ (*as*@*x*#*bs*)) **and** *sublist ys a*
 **shows** (∃ *a*∈*set* (*rev acc* @ *as* @ [*x*]). *sublist ys a*) ∨ *sublist ys* (*concat bs* @ *cs*)
**proof**(*cases a* ∈ *set* (*rev acc* @ *as* @ [*x*]))
 **case** *True*
 **then show** *?thesis* **using** *assms*(*2*) **by** *blast*
**next**
 **case** *False*
 **then have** *a* ∈ *set bs* **using** *assms*(*1*) **by** *simp*
 **then show** *?thesis*
  **using** *assms*(*2*) *concat-all-sublist*[*of bs*]
   *sublist-order.dual-order.trans*[**where** *c* = *ys*, **where** *b* = *concat bs*]
  **by** *fastforce*
**qed**

**lemma** *sublist-split-concat'*:
 ∃ *a* ∈ *set* (*acc* @ (*as*@*x*#*bs*)). *sublist ys a* ∨ *sublist ys cs*
  ⟹ (∃ *a*∈*set* (*rev acc* @ *as* @ [*x*]). *sublist ys a*) ∨ *sublist ys* (*concat bs* @ *cs*)
 **using** *sublist-split-concat sublist-app-l*[*of ys cs*] **by** *blast*

**lemma** *make-list-notP*:
 **assumes** *asi rank r cost*
  **and** *r* ∉ *set U*
  **and** *U* ≠ []
  **and** *set* (*as*@*U*@*bs*@*cs*) = ⋃(*set* ‘ *Y*)
  **and** *distinct* (*as*@*U*@*bs*@*cs*)
  **and** *take 1* (*as*@*U*@*bs*@*cs*) = [*r*]
  **and** *forward* (*as*@*U*@*bs*@*cs*)
  **and** *concat* (*rev acc* @ *ys*) = *bs*
  **and** (∀ *xs* ∈ *Y*. *sublist xs as* ∨ *sublist xs U*
    ∨ (∃ *x* ∈ *set* (*acc* @ *ys*). *sublist xs x*) ∨ *sublist xs cs*)
  **and** *rank* (*rev V*) ≤ *rank* (*rev U*)
  **and** ⋀*xs*. ⟦*xs* ∈ *set ys*; ∃ *x* ∈ *set U*. ∃ *y* ∈ *set xs*. *x* →⁺ *T y*⟧
    ⟹ *rank* (*rev V*) ≤ *rank* (*rev xs*)
  **and** *P* = (λ*x*. *rank* (*rev x*) < *rank* (*rev V*))

**and** $\forall xs.$ *fwd-sub r Y xs* $\longrightarrow$ *cost* (*rev* (*as*@*U*@*bs*@*cs*)) $\leq$ *cost* (*rev xs*)
**and** $\forall x \in$ *set ys*. $x \neq []$
**and** $\forall x \in$ *set ys*. *forward x*
**and** $\forall x \in$ *set acc*. $x \neq []$
**and** $\forall x \in$ *set acc*. *forward x*
**and** $\forall x \in$ *set acc*. $\neg P\ x$
**shows** $\forall x \in$*set* (*make-list-P P acc ys*). $\neg P\ x$
**using** *assms* **proof**(*induction P acc ys rule*: *make-list-P.induct*)
**case** (*1 P acc xs*)
**then show** *?case*
**proof**(*cases List.extract P xs*)
**case** *None*
**then have** $\forall x \in$ *set xs*. $\neg$ *P x* **by** (*simp add*: *extract-None-iff*)
**then show** *?thesis* **using** *1.prems*(*18*) *None* **by** *auto*
**next**
**case** (*Some a*)
**then obtain** *as′ x bs′* **where** *x-def*[*simp*]: *a* = (*as′*,*x*,*bs′*) **by**(*cases a*) *auto*
**then have** *x*: $\forall x \in$ *set* (*rev as′* @ *acc*). $\neg P\ x\ xs$ = *as′*@*x*#*bs′ rank* (*rev x*) < *rank* (*rev V*)
**using** *Some 1.prems*(*12*,*18*) **by** (*auto simp*: *extract-Some-iff*)
**have** $x \neq []$ **using** *1.prems*(*14*) *Some* **by** (*simp add*: *extract-Some-iff*)
**have** *eq*: *as*@*U*@*bs*@*cs* = *as*@*U*@(*concat* (*rev acc* @ *as′* @ [*x*])) @ (*concat bs′* @ *cs*)
**using** *1.prems*(*8*) *Some* **by** (*simp add*: *extract-Some-iff*)
**then have** *0*: *set* (*as*@*U*@(*concat* (*rev acc* @ *as′* @ [*x*])) @ (*concat bs′* @ *cs*)) = $\bigcup$ (*set* ‘ *Y*)
**using** *1.prems*(*4*) **by** *argo*
**have** *2*: *distinct* (*as*@*U*@(*concat* (*rev acc* @ *as′* @ [*x*])) @ (*concat bs′* @ *cs*))
**using** *1.prems*(*5*) *eq* **by** *argo*
**have** *3*: *take 1* (*as*@*U*@(*concat* (*rev acc* @ *as′* @ [*x*])) @ (*concat bs′* @ *cs*)) = [*r*]
**using** *1.prems*(*6*) *eq* **by** *argo*
**have** *4*: *forward* (*as*@*U*@(*concat* (*rev acc* @ *as′* @ [*x*])) @ (*concat bs′* @ *cs*))
**using** *1.prems*(*7*) *eq* **by** *argo*
**have** *5*: *concat* (*rev* (*rev as′* @ *acc*) @ [*x*]) = *concat* (*rev acc* @ *as′* @ [*x*]) **by** *simp*
**have** *6*: $\forall xs \in Y$. *sublist xs as* $\vee$ *sublist xs U*
$\vee$ ($\exists x \in$*set* ((*rev as′* @ *acc*) @ [*x*]). *sublist xs x*) $\vee$ *sublist xs* (*concat bs′* @ *cs*)
**using** *1.prems*(*9*) *x*(*2*) *sublist-split-concat′*[*of acc as′ x bs′*, **where** *cs* = *cs*]
**by** *auto*
**have** *7*: $\neg$ ($\exists x' \in$*set U*. $\exists y \in$*set x*. $x' \rightarrow^+_T y$) **using** *1.prems*(*11*) *x*(*2*,*3*) **by** *fastforce*
**have** *8*: $\forall xs$. *fwd-sub r Y xs*
$\longrightarrow$ *cost* (*rev* (*as*@*U*@*concat*(*rev acc*@*as′*@[*x*])@*concat bs′*@*cs*)) $\leq$ *cost* (*rev xs*)
**using** *1.prems*(*13*) *eq* **by** *simp*
**have** *notP*: $\forall x \in$*set* (*combine-lists-P P x* (*rev as′* @ *acc*)). $\neg$ *P x* $\wedge$ *forward x*
**using** *1.prems*(*14*−*17*) *x*(*2*)

*combine-lists-notP*[*OF* *1*(*2*) ‹*x*≠[]› *1*(*3*,*4*) *0* *2* *3* *4* *5* *6* *1*(*11*) *7* *x*(*3*) *1*(*13*) *x*(*1*) *8*]
**by** *auto*
**have** *cnct*: *concat* (*rev* (*combine-lists-P P x* (*rev as'* @ *acc*)) @ *bs'*) = *bs*
**using** *1.prems*(*8*) *combine-lists-concat-rev-eq*[*of P*] *x*(*2*) **by** *simp*
**have** *sblst*: ∀ *xs*∈*Y*. *sublist xs as* ∨ *sublist xs U*
∨ (∃ *a*∈*set* (*combine-lists-P P x* (*rev as'* @ *acc*) @ *bs'*). *sublist xs a*) ∨ *sublist xs cs*
**using** *1.prems*(*9*) *x*(*2*) *combine-lists-sublists*[*of x rev as'*@*acc*, **where** *P*=*P*] **by** *auto*
**have** ∀ *x*∈*set* (*combine-lists-P P x* (*rev as'* @ *acc*)). *x* ≠ []
**using** *combine-lists-nempty*[*of rev as'* @ *acc*] *1.prems*(*14*,*16*) *x*(*2*) **by** *auto*
**then have** ∀ *x*∈*set* (*make-list-P P* (*combine-lists-P P x* (*rev as'* @ *acc*)) *bs'*). ¬ *P x*
**using** *1.IH*[*OF Some x-def*[*symmetric*] *refl* *1*(*2*−*8*) *cnct sblst* *1*(*11*−*14*)] *notP* *x*(*2*) *1*(*15*,*16*)
**by** *simp*
**then show** *?thesis* **using** *Some* **by** *simp*
**qed**
**qed**

**lemma** *no-back-reach1-if-fwd-dstct-bs*:
⟦*forward* (*as*@*concat bs*@*V*@*cs*); *distinct* (*as*@*concat bs*@*V*@*cs*); *xs* ∈ *set bs*⟧
⟹ ¬(∃ *x'*∈*set V*. ∃ *y*∈*set xs*. *x'* →⁺ *T y*)
**using** *no-back-reach1-if-fwd-dstct*[*of as*@*concat bs V*@*cs*] **by** *auto*

**lemma** *mid-ranks-ge-if-reach1*:
**assumes** [] ∉ *Y*
**and** *U* ∈ *Y*
**and** *distinct* (*as*@*U*@*bs*@*V*@*cs*)
**and** *forward* (*as*@*U*@*bs*@*V*@*cs*)
**and** *concat bs'* = *bs*
**and** *concat cs'* = *cs*
**and** *set* (*as'*@*U*#*bs'*@*V*#*cs'*) = *Y*
**and** ⋀*xs*. ⟦*xs* ∈ *Y*; ∃ *y*∈*set xs*. ¬(∃ *x'*∈*set V*. *x'* →⁺ *T y*) ∧ (∃ *x*∈*set U*. *x* →⁺ *T y*); *xs* ≠ *U*⟧
⟹ *rank* (*rev V*) ≤ *rank* (*rev xs*)
**shows** ∀ *xs* ∈ *set bs'*. (∃ *x*∈*set U*. ∃ *y*∈*set xs*. *x* →⁺ *T y*) ⟶ *rank* (*rev V*) ≤ *rank* (*rev xs*)
**proof** −
**have** ∀ *xs* ∈ *set bs'*. ∀ *y*∈*set xs*. ¬(∃ *x*∈*set V*. *x* →⁺ *T y*)
**using** *assms*(*3*−*6*) *no-back-reach1-if-fwd-dstct-bs*[*of as*@*U*] **by** *fastforce*
**then have** *0*: ∀ *xs* ∈ *set bs'*. (∃ *y*∈*set xs*. ∃ *x*∈*set U*. *x* →⁺ *T y*)
⟶ (∃ *y*∈*set xs*. ∃ *x*∈*set U*. ¬ (∃ *x'*∈*set V*. *x'* →⁺ *T y*) ∧ *x* →⁺ *T y*)
**by** *blast*
**have** ∀ *xs* ∈ *set bs'*. *xs* ≠ *U*
**using** *assms*(*1*−*3*,*5*) *concat-all-sublist empty-if-sublist-dsjnt*[*of U U*] **by** *fastforce*
**then have** ⋀*xs*. ⟦*xs* ∈ *set bs'*; ∃ *y*∈*set xs*. ∃ *x*∈*set U*. *x* →⁺ *T y*⟧

$\Longrightarrow xs \neq U \wedge (\exists\, y{\in}set\ xs.\ \exists\, x{\in}set\ U.\ \neg\ (\exists\, x'{\in}set\ V.\ x' \rightarrow^+_T y) \wedge x \rightarrow^+_T y) \wedge xs \in Y$
**using** *0 assms(7)* **by** *auto*
**then show** *?thesis* **using** *assms(8)* **by** *blast*
**qed**

**lemma** *bs-ranks-only-ge*:
**assumes** *asi rank r cost*
**and** $\forall\, xs \in Y.$ *forward xs*
**and** $[] \notin Y$
**and** $r \notin set\ U$
**and** $U \in Y$
**and** *set* $(as@U@bs@V@cs) = \bigcup (set\ `\ Y)$
**and** *distinct* $(as@U@bs@V@cs)$
**and** *take 1* $(as@U@bs@V@cs) = [r]$
**and** *forward* $(as@U@bs@V@cs)$
**and** *concat* $as' = as$
**and** *concat* $bs' = bs$
**and** *concat* $cs' = cs$
**and** *set* $(as'@U\#bs'@V\#cs') = Y$
**and** *rank* $(rev\ V) \leq rank\ (rev\ U)$
**and** $\forall$ *zs. fwd-sub r Y zs* $\longrightarrow$ *cost* $(rev\ (as@U@bs@V@cs)) \leq cost\ (rev\ zs)$
**and** $\bigwedge xs.\ [\![xs \in Y;\ \exists\, y{\in}set\ xs.\ \neg(\exists\, x'{\in}set\ V.\ x' \rightarrow^+_T y) \wedge (\exists\, x{\in}set\ U.\ x \rightarrow^+_T y);\ xs \neq U]\!]$
$\Longrightarrow rank\ (rev\ V) \leq rank\ (rev\ xs)$
**shows** $\exists\, zs.$ *concat zs = bs* $\wedge\ (\forall\, z \in set\ zs.\ rank\ (rev\ V) \leq rank\ (rev\ z)) \wedge [] \notin set\ zs$
**proof** −
**let** *?P* $= \lambda x.\ rank\ (rev\ x) < rank\ (rev\ V)$
**have** $U \neq []$ **using** *assms(3,5)* **by** *blast*
**have** *cnct*: *concat* $(rev\ [] @ bs') = bs$ **using** *assms(11)* **by** *simp*
**have** $\forall\, xs{\in}Y.$ *sublist xs as* $\vee$ $xs = U \vee xs = V$
$\vee\ (\exists\, x{\in}set\ ([] @ bs').$ *sublist xs x*$) \vee$ *sublist xs cs*
**using** *assms(10,12,13) concat-all-sublist* **by** *auto*
**then have** *sblst*:
$\forall\, xs{\in}Y.$ *sublist xs as* $\vee$ *sublist xs U* $\vee\ (\exists\, x{\in}set\ ([] @ bs').$ *sublist xs x*$) \vee$ *sublist xs* $(V@cs)$
**using** *sublist-app-l* **by** *fast*
**have** *0*: $\bigwedge xs.\ [\![xs \in set\ bs';\ \exists\, x{\in}set\ U.\ \exists\, y{\in}set\ xs.\ x \rightarrow^+_T y]\!] \Longrightarrow rank\ (rev\ V) \leq rank\ (rev\ xs)$
**using** *mid-ranks-ge-if-reach1*[*OF assms(3,5,7,9,11−13)*] *assms(16)* **by** *blast*
**have** $\forall\, x{\in}set\ bs'.\ x \neq []$ **using** *assms(3,13)* **by** *auto*
**moreover have** *2*: $\forall\, x{\in}set\ bs'.$ *forward x* **using** *assms(2,13)* **by** *auto*
**ultimately have** $(\forall\, x{\in}set\ (make\text{-}list\text{-}P\ ?P\ []\ bs').\ rank\ (rev\ V) \leq rank\ (rev\ x))$
**using** *assms(15)*
*make-list-notP*[*OF assms(1,4)* ‹$U \neq []$› *assms(6−9) cnct sblst assms(14) 0 refl*]
**by** *fastforce*
**then show** *?thesis*

**using** *assms*(3,11,13) *make-list-concat-rev-eq*[*of ?P* []] *make-list-nempty*[*of* [] *bs'*] **by** *auto*
**qed**

**lemma** *cost-ge-if-all-bs-ge*:
**assumes** *asi rank r cost*
**and** $V \neq []$
**and** *distinct* (*as*@*ds*@*concat bs*@$V$@*cs*)
**and** *take 1 as* = [*r*]
**and** *forward V*
**and** $\forall z \in set\ bs.\ rank\ (rev\ V) \leq rank\ (rev\ z)$
**and** $[] \notin set\ bs$
**shows** *cost* (*rev* (*as*@*ds*@$V$@*concat bs*@*cs*)) ≤ *cost* (*rev* (*as*@*ds*@*concat bs*@$V$@*cs*))
**using** *assms* **proof**(*induction bs arbitrary*: *ds*)
**case** (*Cons b bs*)
**have** *0*: *distinct* (*as*@(*ds*@*b*)@*concat bs*@$V$@*cs*) **using** *Cons.prems*(3) **by** *simp*
**have** *r-b*: *rank* (*rev V*) ≤ *rank* (*rev b*) **using** *Cons.prems*(6) **by** *simp*
**have** $b \neq []$ **using** *Cons.prems*(7) **by** *auto*
**have** *dst*: *distinct* ((*as*@*ds*)@$V$@*b*@*concat bs*@*cs*) **using** *Cons.prems*(3) **by** *auto*
**have** *take 1* ((*as*@*ds*)@$V$@*b*@*concat bs*@*cs*) = [*r*]
**using** *Cons.prems*(4) *take1-singleton-app* **by** *metis*
**moreover have** *take 1* ((*as*@*ds*)@*b*@$V$@*concat bs*@*cs*) = [*r*]
**using** *Cons.prems*(4) *take1-singleton-app* **by** *metis*
**ultimately have** *cost* (*rev* (*as*@*ds*@$V$@*b*@*concat bs*@*cs*)) ≤ *cost* (*rev* (*as*@*ds*@*b*@$V$@*concat bs*@*cs*))
**using** *asi-le-rfst*[*OF Cons.prems*(1) *r-b Cons.prems*(2) ‹*b*≠[]› *dst*] **by** *simp*
**then show** *?case* **using** *Cons.IH*[*OF Cons.prems*(1,2) *0*] *Cons.prems*(4−7) **by** *simp*
**qed**(*simp*)

**lemma** *bs-ge-if-all-ge*:
**assumes** *asi rank r cost*
**and** $V \neq []$
**and** *distinct* (*as*@*bs*@$V$@*cs*)
**and** *take 1 as* = [*r*]
**and** *forward V*
**and** *concat bs'* = *bs*
**and** $\forall z \in set\ bs'.\ rank\ (rev\ V) \leq rank\ (rev\ z)$
**and** $[] \notin set\ bs'$
**and** $bs \neq []$
**shows** *rank* (*rev V*) ≤ *rank* (*rev bs*)
**proof** −
**have** *dst*: *distinct* (*as*@[]@*concat bs'*@$V$@*cs*) **using** *assms*(3,6) **by** *simp*
**then have** *cost-le*: *cost* (*rev* (*as*@$V$@*bs*@*cs*)) ≤ *cost* (*rev* (*as*@*bs*@$V$@*cs*))
**using** *cost-ge-if-all-bs-ge*[*OF assms*(1,2) *dst*] *assms*(3−9) **by** *simp*
**have** *tk1*: *take 1* ((*as*)@*bs*@$V$@*cs*) = [*r*] **using** *assms*(4) *take1-singleton-app* **by** *metis*
**have** *tk1'*: *take 1* ((*as*)@$V$@*bs*@*cs*) = [*r*] **using** *assms*(4) *take1-singleton-app* **by** *metis*

**have** *dst*: *distinct* ((*as*)@*V*@*bs*@*cs*) **using** *assms*(*3*) **by** *auto*
**show** *?thesis* **using** *asi-le-iff-rfst*[*OF assms*(*1*,*2*,*9*) *tk1' tk1 dst*] *cost-le* **by** *simp*
**qed**

**lemma** *bs-ge-if-optimal*:
**assumes** *asi rank r cost*
**and** ∀ *xs* ∈ *Y*. ∀ *ys* ∈ *Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}
**and** ∀ *xs* ∈ *Y*. *forward xs*
**and** [] ∉ *Y*
**and** *finite Y*
**and** *r* ∉ *set U*
**and** *U* ∈ *Y*
**and** *V* ∈ *Y*
**and** *distinct* (*as*@*U*@*bs*@*V*@*cs*)
**and** *set* (*as*@*U*@*bs*@*V*@*cs*) = ⋃(*set* ‘ *Y*)
**and** ∀ *xs* ∈ *Y*. *sublist xs* (*as*@*U*@*bs*@*V*@*cs*)
**and** *take 1* (*as*@*U*@*bs*@*V*@*cs*) = [*r*]
**and** *forward* (*as*@*U*@*bs*@*V*@*cs*)
**and** *bs* ≠ []
**and** *rank* (*rev V*) ≤ *rank* (*rev U*)
**and** ∀ *zs*. *fwd-sub r Y zs* ⟶ *cost* (*rev* (*as*@*U*@*bs*@*V*@*cs*)) ≤ *cost* (*rev zs*)
**and** ⋀*xs*. ⟦*xs* ∈ *Y*; ∃ *y*∈*set xs*. ¬(∃ *x'*∈*set V*. *x'* →⁺ *T y*) ∧ (∃ *x*∈*set U*. *x* →⁺ *T y*); *xs* ≠ *U*⟧
⟹ *rank* (*rev V*) ≤ *rank* (*rev xs*)
**shows** *rank* (*rev V*) ≤ *rank* (*rev bs*)
**proof** −
**obtain** *as' bs' cs'* **where** *bs'-def*: *concat as'* = *as concat bs'* = *bs concat cs'* = *cs*
*set* (*as'*@*U*#*bs'*@*V*#*cs'*) = *Y*
**using** *concat-split-UV*[*OF assms*(*2*,*5*,*7*−*11*)] *assms*(*4*,*7*,*8*) **by** *blast*
**obtain** *bs2* **where** *bs2-def*:
*concat bs2* = *bs* (∀ *z*∈*set bs2*. *rank* (*rev V*) ≤ *rank* (*rev z*)) [] ∉ *set bs2*
**using** *bs-ranks-only-ge*[*OF assms*(*1*,*3*,*4*,*6*,*7*,*10*,*9*,*12*,*13*) *bs'-def assms*(*15*−*17*)]
**by** *blast*
**have** *V* ≠ [] **using** *assms*(*4*,*8*) **by** *blast*
**have** *take 1 as* = [*r*] **using** *take1-split-nelem-nempty*[*OF assms*(*12*)] *assms*(*4*,*6*,*7*)
**by** *blast*
**then have** *take 1* (*as*@*U*) = [*r*] **using** *take1-singleton-app* **by** *fast*
**then show** *?thesis*
**using** *bs-ge-if-all-ge*[*OF assms*(*1*) ‹*V*≠[]›, *of as*@*U*] *bs2-def assms*(*3*,*8*,*9*,*14*)
**by** *auto*
**qed**

**lemma** *bs-ranks-only-ge-r*:
**assumes** [] ∉ *Y*
**and** *distinct* (*as*@*U*@*bs*@*V*@*cs*)
**and** *forward* (*as*@*U*@*bs*@*V*@*cs*)
**and** *as* = []
**and** *concat bs'* = *bs*
**and** *concat cs'* = *cs*

**and** *set* (*U*#*bs*′@*V*#*cs*′) = *Y*
**and** ⋀*xs*. ⟦*xs* ∈ *Y*; ∃ *y*∈*set xs*. ¬(∃ *x*′∈*set V*. *x*′ →⁺$_T$ *y*) ∧ (∃ *x*∈*set U*. *x* →⁺$_T$ *y*); *xs* ≠ *U*⟧
⟹ *rank* (*rev V*) ≤ *rank* (*rev xs*)
**shows** ∀ *z* ∈ *set bs*′. *rank* (*rev V*) ≤ *rank* (*rev z*)
**proof** −
**have** *U* ∈ *Y* **using** *assms*(7) **by** *auto*
**then have** *U* ≠ [] **using** *assms*(1) **by** *blast*
**have** *V* ≠ [] **using** *assms*(1,7) **by** *auto*
**have** *0*: ⋀*xs*. ⟦*xs* ∈ *set bs*′; ∃ *x*∈*set U*. ∃ *y*∈*set xs*. *x* →⁺$_T$ *y*⟧ ⟹ *rank* (*rev V*) ≤ *rank* (*rev xs*)
**using** *mid-ranks-ge-if-reach1*[*OF assms*(1) ‹*U*∈*Y*› *assms*(2,3,5,6), *of* []] *assms*(7,8)
**by** *auto*
**have** ∃ *x y ys*. *x*#*y*#*ys*= *as*@*U*@*bs*@*V*@*cs*
**using** ‹*U*≠[]› ‹*V*≠[]› *append-Cons append.left-neutral list.exhaust* **by** *metis*
**then have** *hd-T*: *hd* (*as*@*U*@*bs*@*V*@*cs*) ∈ *verts T* **using** *hd-in-verts-if-forward assms*(3) **by** *metis*
**moreover have** ∀ *x*∈*set bs*′. ∀ *y*∈*set x*. *y* ∈ *set* (*as*@*U*@*bs*@*V*@*cs*) **using** *assms*(5)
**by** *auto*
**ultimately have** ∀ *x*∈*set bs*′. ∀ *y*∈*set x*. *hd* (*U*@*bs*@*V*@*cs*) →*$_T$ *y*
**using** *hd-reach-all-forward assms*(3,4) **by** *auto*
**then have** *1*: ∀ *x*∈*set bs*′. ∀ *y*∈*set x*. *hd U* →*$_T$ *y* **using** *assms*(1,7) **by** *auto*
**have** ∀ *x*∈*set bs*′. ∀ *y*∈*set x*. *y* ∉ *set U* **using** *assms*(2,5) **by** *auto*
**then have** ∀ *x*∈*set bs*′. ∀ *y*∈*set x*. *y* ≠ *hd U* **using** *assms*(1,7) **by** *fastforce*
**then have** ∀ *x*∈*set bs*′. ∀ *y*∈*set x*. *hd U* →⁺$_T$ *y* **using** *1* **by** *blast*
**then have** ∀ *x*∈*set bs*′. ∃ *y*∈*set x*. *hd U* →⁺$_T$ *y* **using** *assms*(1,7) **by** *auto*
**then show** *?thesis* **using** *0* ‹*U* ≠ []› *hd-in-set* **by** *blast*
**qed**

**lemma** *bs-ge-if-rU*:
**assumes** *asi rank r cost*
**and** ∀ *xs* ∈ *Y*. ∀ *ys* ∈ *Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}
**and** ∀ *xs* ∈ *Y*. *forward xs*
**and** [] ∉ *Y*
**and** *finite Y*
**and** *r* ∈ *set U*
**and** *U* ∈ *Y*
**and** *V* ∈ *Y*
**and** *distinct* (*as*@*U*@*bs*@*V*@*cs*)
**and** *set* (*as*@*U*@*bs*@*V*@*cs*) = ⋃(*set* ‘ *Y*)
**and** ∀ *xs* ∈ *Y*. *sublist xs* (*as*@*U*@*bs*@*V*@*cs*)
**and** *take 1* (*as*@*U*@*bs*@*V*@*cs*) = [*r*]
**and** *forward* (*as*@*U*@*bs*@*V*@*cs*)
**and** *bs* ≠ []
**and** ⋀*xs*. ⟦*xs* ∈ *Y*; ∃ *y*∈*set xs*. ¬(∃ *x*′∈*set V*. *x*′ →⁺$_T$ *y*) ∧ (∃ *x*∈*set U*. *x* →⁺$_T$ *y*); *xs* ≠ *U*⟧
⟹ *rank* (*rev V*) ≤ *rank* (*rev xs*)
**shows** *rank* (*rev V*) ≤ *rank* (*rev bs*)
**proof** −

**obtain** *as' bs' cs'* **where** *bs'-def*: *concat as' = as concat bs' = bs concat cs' = cs set (as'@U#bs'@V#cs') = Y*
**using** *concat-split-UV*[*OF assms*(2,5,7−11)] *assms*(4,7,8) **by** *blast*
**have** *take 1 U = [r]* **using** *take1-mid-if-elem*[*OF assms*(12,6,9)] **.**
**moreover have** *as = []* **using** *take1-empty-if-mid*[*OF assms*(12,6,9)] **.**
**ultimately have** *tk1*: *take 1 (as@U) = [r]* **by** *simp*
**then have** *set (U#bs'@V#cs') = Y* **using** *bs'-def*(1,4) *assms*(4) ‹*as=[]*› **by** *auto*
**then have** *0*: (∀ *z*∈*set bs'*. *rank (rev V) ≤ rank (rev z)*)
**using** *bs-ranks-only-ge-r*[*OF assms*(4,9,13) ‹*as=[]*› *bs'-def*(2,3)] *assms*(15) **by** *blast*
**have** *V ≠ []* **using** *assms*(4,8) **by** *blast*
**have** *[] ∉ set bs'* **using** *assms*(4) *bs'-def*(2,4) **by** *auto*
**then show** *?thesis*
**using** *bs-ge-if-all-ge*[*OF assms*(1) ‹*V≠[]*›, *of as@U*] *0 bs'-def*(2) *tk1 assms*(3,8,9,14) **by** *auto*
**qed**

**lemma** *sublist-before-if-before*:
**assumes** *hd xs = root* **and** *forward xs* **and** *distinct xs*
**and** *sublist U xs* **and** *sublist V xs* **and** *before U V*
**shows** ∃ *as bs cs*. *as @ U @ bs @ V @ cs = xs*
**proof** (*rule ccontr*)
**assume** ∄ *as bs cs*. *as @ U @ bs @ V @ cs = xs*
**then obtain** *as bs cs* **where** *V-bf-U*: *xs = as @ V @ bs @ U @ cs*
**using** *sublist-behind-if-nbefore*[*OF assms*(4,5)] *assms*(6) *before-def* **by** *blast*
**obtain** *x y* **where** *x-def*: *x ∈ set U y ∈ set V x* →$_T$ *y*
**using** *assms*(6) *before-def* **by** *auto*
**then obtain** *i* **where** *i-def*: *V!i = y i < length V* **by** (*auto simp*: *in-set-conv-nth*)
**then have** *i-xs*: (*as@V@bs@U@cs*)!(*i + length as*) = *y* **by** (*simp add*: *nth-append*)
**have** *root ≠ y* **using** *x-def*(3) *dominated-not-root* **by** *auto*
**then have** *i + length as > 0* **using** *i-def*(2) *i-xs assms*(1,5) *V-bf-U hd-conv-nth*[*of xs*] **by** *force*
**then have** *i + length as ≥ 1* **by** *linarith*
**then have** *i + length as ∈* {*1..length* (*as@V@bs@U@cs*) − *1*} **using** *i-def*(2) **by** *simp*
**then obtain** *j* **where** *j-def*: *j < i + length as* (*as@V@bs@U@cs*)!*j* →$_T$ *y*
**using** *assms*(2) *V-bf-U i-xs* **unfolding** *forward-def* **by** *blast*
**then have** (*as@V@bs@U@cs*)!*j* = (*as@V*)!*j* **using** *i-def*(2) **by** (*auto simp*: *nth-append*)
**then have** (*as@V@bs@U@cs*)!*j ∈ set* (*as@V*) **using** *i-def*(2) *j-def*(1) *nth-mem*[*of j as@V*] **by** *simp*
**then have** (*as@V@bs@U@cs*)!*j ≠ x* **using** *assms*(3) *V-bf-U x-def*(1) **by** *auto*
**then show** *False* **using** *j-def*(2) *x-def*(3) *two-in-arcs-contr* **by** *fastforce*
**qed**

**lemma** *forward-UV-lists-subset*:
{*x*. *set x = X ∧ distinct x ∧ take 1 x = [r] ∧ forward x ∧* (∀ *xs ∈ Y*. *sublist xs x*)}

$\subseteq \{x.\ set\ x = X \wedge distinct\ x\}$
**by** *blast*

**lemma** *forward-UV-lists-finite*:
*finite xs*
$\Longrightarrow$ *finite* $\{x.\ set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r] \wedge forward\ x \wedge (\forall\, xs \in Y.\ sublist\ xs\ x)\}$
**using** *distinct-seteq-finite finite-subset*[*OF forward-UV-lists-subset*] **by** *auto*

**lemma** *forward-UV-lists-arg-min-ex-aux*:
⟦*finite ys*; $ys \neq \{\}$;
$ys = \{x.\ set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r] \wedge forward\ x \wedge (\forall\, xs \in Y.\ sublist\ xs\ x)\}$⟧
$\Longrightarrow \exists\, y \in ys.\ \forall\, z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
**using** *arg-min-if-finite*(*1*)[*of ys f*] *arg-min-least*[*of ys*, **where** *?f* = *f*] **by** *auto*

**lemma** *forward-UV-lists-arg-min-ex*:
⟦*finite xs*; $ys \neq \{\}$;
$ys = \{x.\ set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r] \wedge forward\ x \wedge (\forall\, xs \in Y.\ sublist\ xs\ x)\}$⟧
$\Longrightarrow \exists\, y \in ys.\ \forall\, z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
**using** *forward-UV-lists-finite forward-UV-lists-arg-min-ex-aux* **by** *auto*

**lemma** *forward-UV-lists-argmin-ex'*:
**fixes** $f :: 'a\ list \Rightarrow real$
**assumes** $P = (\lambda x.\ set\ x = X \wedge distinct\ x \wedge take\ 1\ x = [r])$
**and** $Q = (\lambda ys.\ P\ ys \wedge forward\ ys \wedge (\forall\, xs \in Y.\ sublist\ xs\ ys))$
**and** $\exists\, x.\ Q\ x$
**shows** $\exists\, zs.\ Q\ zs \wedge (\forall\, as.\ Q\ as \longrightarrow f\ zs \leq f\ as)$
**using** *forward-UV-lists-arg-min-ex*[*of X* $\{x.\ Q\ x\}$] **using** *assms* **by** *fastforce*

**lemma** *forward-UV-lists-argmin-ex*:
**fixes** $f :: 'a\ list \Rightarrow real$
**assumes** $\exists\, x.\ fwd\text{-}sub\ r\ Y\ x$
**shows** $\exists\, zs.\ fwd\text{-}sub\ r\ Y\ zs \wedge (\forall\, as.\ fwd\text{-}sub\ r\ Y\ as \longrightarrow f\ zs \leq f\ as)$
**using** *forward-UV-lists-argmin-ex' assms* **unfolding** *fwd-sub-def unique-set-r-def* **by** *simp*

**lemma** *no-gap-if-contr-seq-fwd*:
**assumes** *asi rank root cost*
**and** $\forall\, xs \in Y.\ \forall\, ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
**and** $\forall\, xs \in Y.\ forward\ xs$
**and** $[] \notin Y$
**and** *finite Y*
**and** $U \in Y$
**and** $V \in Y$
**and** *before U V*
**and** $rank\ (rev\ V) \leq rank\ (rev\ U)$
**and** $\bigwedge xs.\ ⟦xs \in Y;\ \exists\, y{\in}set\ xs.\ \neg(\exists\, x'{\in}set\ V.\ x' \rightarrow^+_T y) \wedge (\exists\, x{\in}set\ U.\ x$

$\rightarrow^+_T$ *y*); *xs* $\neq$ *U*⟧
⟹ *rank* (*rev V*) ≤ *rank* (*rev xs*)
**and** ∃ *x*. *fwd-sub root Y x*
**shows** ∃ *zs*. *fwd-sub root Y zs* ∧ *sublist* (*U*@*V*) *zs*
∧ (∀ *as*. *fwd-sub root Y as* ⟶ *cost* (*rev zs*) ≤ *cost* (*rev as*))
**proof** −
**obtain** *zs* **where** *zs-def*:
*set zs* = ⋃(*set* ‘ *Y*) *distinct zs take 1 zs* = [*root*] *forward zs*
(∀ *xs* ∈ *Y*. *sublist xs zs*) (∀ *as*. *fwd-sub root Y as* ⟶ *cost* (*rev zs*) ≤ *cost* (*rev as*))
**using** *forward-UV-lists-argmin-ex*[*OF assms*(*11*), *of* λ*xs*. *cost* (*rev xs*)]
**unfolding** *unique-set-r-def fwd-sub-def* **by** *blast*
**then have** *hd zs* = *root* **using** *hd-eq-take1* **by** *fast*
**then obtain** *as bs cs* **where** *bs-def*: *as* @ *U* @ *bs* @ *V* @ *cs* = *zs*
**using** *sublist-before-if-before zs-def*(*2*,*4*,*5*) *assms*(*6*−*8*) **by** *blast*
**then have** *bs-prems*: *distinct* (*as*@*U*@*bs*@*V*@*cs*) *set* (*as*@*U*@*bs*@*V*@*cs*) = ⋃(*set* ‘ *Y*)
∀ *xs*∈*Y*. *sublist xs* (*as*@*U*@*bs*@*V*@*cs*) *take 1* (*as*@*U*@*bs*@*V*@*cs*) = [*root*] *forward* (*as*@*U*@*bs*@*V*@*cs*)
**using** *zs-def*(*1*−*5*) **by** *auto*
**show** *?thesis*
**proof**(*cases bs* = [])
**case** *True*
**then have** *sublist* (*U*@*V*) *zs* **using** *bs-def sublist-def* **by** *force*
**then show** *?thesis* **using** *zs-def* **unfolding** *unique-set-r-def fwd-sub-def* **by** *blast*
**next**
**case** *bs-nempty*: *False*
**then have** *rank-le*: *rank* (*rev V*) ≤ *rank* (*rev bs*)
**proof**(*cases root* ∈ *set U*)
**case** *True*
**then show** *?thesis*
**using** *bs-ge-if-rU*[*OF assms*(*1*−*5*) *True assms*(*6*,*7*) *bs-prems bs-nempty assms*(*10*)]
**by** *blast*
**next**
**case** *False*
**have** ∀ *zs*. *fwd-sub root Y zs* ⟶ *cost* (*rev* (*as*@*U*@*bs*@*V*@*cs*)) ≤ *cost* (*rev zs*)
**using** *zs-def*(*6*) *bs-def* **by** *blast*
**then show** *?thesis*
**using** *bs-ge-if-optimal*[*OF assms*(*1*−*5*)] *bs-nempty bs-prems False assms*(*6*,*7*,*9*,*10*)
**by** *blast*
**qed**
**have** *0*: *distinct* ((*as*@*U*)@*V*@*bs*@*cs*) **using** *bs-def zs-def*(*2*) **by** *auto*
**have** *take 1* (*as*@*U*) = [*root*]
**using** *bs-def assms*(*4*,*6*) *take1-split-nempty*[*of U as*] *zs-def*(*3*) **by** *fastforce*
**then have** *1*: *take 1* (*as*@*U*@*V*@*bs*@*cs*) = [*root*]
**using** *take1-singleton-app*[*of as*@*U root V*@*bs*@*cs*] **by** *simp*
**have** *2*: ∀ *xs*∈*Y*. *sublist xs* (*as*@*U*@*V*@*bs*@*cs*)

**using** *zs-def*(*5*) *bs-def sublists-preserv-move-VY-all*[*OF assms*(*2*,*6*,*7*)] *assms*(*4*,*6*)
**by** *blast*
**have** $V \neq []$ **using** *assms*(*4*,*7*) **by** *blast*
**have** *cost* (*rev* (*as*@*U*@*V*@*bs*@*cs*)) ≤ *cost* (*rev zs*)
**using** *asi-le-rfst*[*OF assms*(*1*) *rank-le* ‹*V*≠[]› *bs-nempty 0*] *1 zs-def*(*3*) *bs-def*
**by** *simp*
**then have** *cost-le*: ∀ *ys*. *fwd-sub root Y ys* ⟶ *cost* (*rev* (*as*@*U*@*V*@*bs*@*cs*))
≤ *cost* (*rev ys*)
**using** *zs-def*(*6*) **by** *fastforce*
**have** *forward* (*as*@*U*@*V*@*bs*@*cs*)
**using** *move-mid-backward-if-noarc assms*(*8*) *zs-def*(*4*) *bs-def* **by** *blast*
**moreover have** *set* (*as*@*U*@*V*@*bs*@*cs*) = ⋃ (*set* ‘ *Y*)
**unfolding** *zs-def*(*1*)[*symmetric*] *bs-def*[*symmetric*] **by** *force*
**ultimately have** *fwd-sub root Y* (*as*@*U*@*V*@*bs*@*cs*)
**unfolding** *unique-set-r-def fwd-sub-def* **using** *0 1 2* **by** *fastforce*
**moreover have** *sublist* (*U*@*V*) (*as*@*U*@*V*@*bs*@*cs*) **unfolding** *sublist-def* **by**
*fastforce*
**ultimately show** *?thesis* **using** *cost-le* **by** *blast*
**qed**
**qed**

**lemma** *combine-union-sets-alt*:
**fixes** *X Y*
**defines** *Z* ≡ *X* ∪ {*x*. *x* ∈ *Y* ∧ *set x* ∩ ⋃(*set* ‘ *X*) = {}}
**assumes** ∀ *xs* ∈ *Y*. ∀ *ys* ∈ *Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}
**and** ∀ *xs* ∈ *X*. ∀ *ys* ∈ *X*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}
**shows** *Z* = *X* ∪ (*Y* − {*x*. *set x* ∩ ⋃(*set* ‘ *X*) ≠ {}})
**unfolding** *assms*(*1*) **using** *assms*(*2*,*3*) **by** *fast*

**lemma** *combine-union-sets-disjoint*:
**fixes** *X Y*
**defines** *Z* ≡ *X* ∪ {*x*. *x* ∈ *Y* ∧ *set x* ∩ ⋃(*set* ‘ *X*) = {}}
**assumes** ∀ *xs* ∈ *Y*. ∀ *ys* ∈ *Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}
**and** ∀ *xs* ∈ *X*. ∀ *ys* ∈ *X*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}
**shows** ∀ *xs* ∈ *Z*. ∀ *ys* ∈ *Z*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}
**unfolding** *Z-def* **using** *assms*(*2*,*3*) **by** *force*

**lemma** *combine-union-sets-set-sub1-aux*:
**assumes** ∀ *xs* ∈ *Y*. ∀ *ys* ∈ *Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}
**and** ∀ *ys* ∈ *X*. ∃ *U* ∈ *Y*. ∃ *V* ∈ *Y*. *U*@*V* = *ys*
**and** *x* ∈ ⋃(*set* ‘ *Y*)
**shows** *x* ∈ ⋃(*set* ‘ (*X* ∪ {*x*. *x* ∈ *Y* ∧ *set x* ∩ ⋃(*set* ‘ *X*) = {}}))
**proof** −
**let** *?Z* = *X* ∪ {*x*. *x* ∈ *Y* ∧ *set x* ∩ ⋃(*set* ‘ *X*) = {}}
**obtain** *ys* **where** *ys-def*: *x* ∈ *set ys ys* ∈ *Y* **using** *assms*(*3*) **by** *blast*
**then show** *?thesis*
**proof**(*cases ys* ∈ {*x*. *x* ∈ *Y* ∧ *set x* ∩ ⋃(*set* ‘ *X*) = {}})
**case** *True*
**then show** *?thesis* **using** *ys-def*(*1*) **by** *auto*

**next**
**case** *False*
**then obtain** *U V* **where** *U-def*: $U \in Y$ $V \in Y$ $U@V \in X$ *set ys* $\cap$ *set* $(U@V) \neq \{\}$
**using** *ys-def*(*2*) *assms*(*2*) **by** *fast*
**then consider** *set ys* $\cap$ *set U* $\neq \{\}$ | *set ys* $\cap$ *set V* $\neq \{\}$ **by** *fastforce*
**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then have** $U = ys$ **using** *assms*(*1*) *U-def*(*1*) *ys-def*(*2*) **by** *blast*
**then show** *?thesis* **using** *ys-def*(*1*) *U-def*(*3*) **by** *fastforce*
**next**
**case** *2*
**then have** $V = ys$ **using** *assms*(*1*) *U-def*(*2*) *ys-def*(*2*) **by** *blast*
**then show** *?thesis* **using** *ys-def*(*1*) *U-def*(*3*) **by** *fastforce*
**qed**
**qed**
**qed**

**lemma** *combine-union-sets-set-sub1*:
**assumes** $\forall xs \in Y.\ \forall ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
**and** $\forall ys \in X.\ \exists U \in Y.\ \exists V \in Y.\ U@V = ys$
**shows** $\bigcup(set\ `\ Y) \subseteq \bigcup(set\ `\ (X \cup \{x.\ x \in Y \wedge set\ x \cap \bigcup(set\ `\ X) = \{\}\}))$
**using** *combine-union-sets-set-sub1-aux*[*OF assms*] **by** *blast*

**lemma** *combine-union-sets-set-sub2*:
**assumes** $\forall ys \in X.\ \exists U \in Y.\ \exists V \in Y.\ U@V = ys$
**shows** $\bigcup(set\ `\ (X \cup \{x.\ x \in Y \wedge set\ x \cap \bigcup(set\ `\ X) = \{\}\})) \subseteq \bigcup(set\ `\ Y)$
**using** *assms* **by** *fastforce*

**lemma** *combine-union-sets-set-eq*:
**assumes** $\forall xs \in Y.\ \forall ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
**and** $\forall ys \in X.\ \exists U \in Y.\ \exists V \in Y.\ U@V = ys$
**shows** $\bigcup(set\ `\ (X \cup \{x.\ x \in Y \wedge set\ x \cap \bigcup(set\ `\ X) = \{\}\})) = \bigcup(set\ `\ Y)$
**using** *combine-union-sets-set-sub1*[*OF assms*] *combine-union-sets-set-sub2*[*OF assms*(*2*)] **by** *blast*

**lemma** *combine-union-sets-sublists*:
**assumes** *sublist x ys*
**and** $\forall xs \in X \cup \{x.\ x \in Y \wedge set\ x \cap \bigcup(set\ `\ X) = \{\}\}.$ *sublist xs ys*
**and** *xs* $\in$ *insert x X* $\cup \{xs.\ xs \in Y \wedge set\ xs \cap \bigcup(set\ `\ (insert\ x\ X)) = \{\}\}$
**shows** *sublist xs ys*
**using** *assms* **by** *auto*

**lemma** *combine-union-sets-optimal-cost*:
**assumes** *asi rank root cost*
**and** $\forall xs \in Y.\ \forall ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
**and** $\forall xs \in Y.$ *forward xs*
**and** $[] \notin Y$

**and** *finite Y*
**and** $\exists x.$ *fwd-sub root Y x*
**and** $\forall ys \in X. \exists U \in Y. \exists V \in Y.$ *U@V = ys* $\wedge$ *before U V* $\wedge$ *rank (rev V) ≤ rank (rev U)*
$\wedge$ $(\forall xs \in Y. (\exists y{\in}set\ xs. \neg(\exists x'{\in}set\ V. x' \rightarrow^+_T y) \wedge (\exists x{\in}set\ U. x \rightarrow^+_T y) \wedge xs \neq U)$
$\longrightarrow$ *rank (rev V) ≤ rank (rev xs)*)
**and** $\forall xs \in X. \forall ys \in X.$ *xs = ys* $\vee$ *set xs* $\cap$ *set ys = {}*
**and** $\forall xs \in X. \forall ys \in X.$ *xs = ys* $\vee \neg(\exists x{\in}set\ xs. \exists y{\in}set\ ys.\ x \rightarrow^+_T y)$
**and** *finite X*
**shows** $\exists zs.$ *fwd-sub root* $(X \cup \{x.\ x \in Y \wedge set\ x \cap \bigcup(set\ `\ X) = \{\}\})$ *zs*
$\wedge$ $(\forall as.$ *fwd-sub root Y as* $\longrightarrow$ *cost (rev zs) ≤ cost (rev as))*
**using** *assms*(10,1−9) **proof**(*induction X rule*: *finite-induct*)
**case** *empty*
**then show** *?case* **using** *forward-UV-lists-argmin-ex* **by** *simp*
**next**
**case** (*insert x X*)
**let** *?Y = X* $\cup$ $\{xs.\ xs \in Y \wedge set\ xs \cap \bigcup(set\ `\ X) = \{\}\}$
**let** *?X = insert x X* $\cup$ $\{xs.\ xs \in Y \wedge set\ xs \cap \bigcup(set\ `\ (insert\ x\ X)) = \{\}\}$
**obtain** *zs* **where** *zs-def*:
*fwd-sub root ?Y zs* $(\forall as.$ *fwd-sub root Y as* $\longrightarrow$ *cost (rev zs) ≤ cost (rev as))*
**using** *insert.IH*[*OF insert*(4−9)] *insert.prems*(7,8,9) **by** *auto*
**obtain** *U V* **where** *U-def*: *U ∈ Y V ∈ Y U@V = x before U V rank (rev V) ≤ rank (rev U)*
$\forall xs \in Y. (\exists y{\in}set\ xs. \neg(\exists x'{\in}set\ V. x' \rightarrow^+_T y) \wedge (\exists x{\in}set\ U. x \rightarrow^+_T y) \wedge xs \neq U)$
$\longrightarrow$ *rank (rev V) ≤ rank (rev xs)*
**using** *insert.prems*(7) **by** *auto*
**then have** *U*: *U ∈ ?Y* **using** *insert.prems*(2,8) *insert.hyps*(2) **by** *fastforce*
**have** *V*: *V ∈ ?Y* **using** *U-def*(2,3) *insert.prems*(8) *insert.hyps*(2) **by** *fastforce*
**have** *disj*: $\forall xs \in ?Y. \forall ys \in ?Y.$ *xs = ys* $\vee$ *set xs* $\cap$ *set ys = {}*
**using** *combine-union-sets-disjoint*[*of Y X*] *insert.prems*(2,8) **by** *blast*
**have** *fwd*: $\forall xs \in ?Y.$ *forward xs*
**using** *insert.prems*(3,7) *seq-conform-alt seq-conform-if-before* **by** *fastforce*
**have** *nempty*: $[] \notin ?Y$ **using** *insert.prems*(4,7) **by** *blast*
**have** *fin*: *finite ?Y* **using** *insert.prems*(5) *insert.hyps*(1) **by** *simp*
**have** *0*: $\bigwedge xs.$ $[\![xs \in ?Y; \exists y{\in}set\ xs. \neg\ (\exists x'{\in}set\ V. x' \rightarrow^+_T y) \wedge (\exists x{\in}set\ U. x \rightarrow^+_T y); xs \neq U]\!]$
$\Longrightarrow$ *rank (rev V) ≤ rank (rev xs)*
**using** *U-def*(3,6) *insert.prems*(9) *insert.hyps*(2) **by** *auto*
**then have** $\exists zs.$ *fwd-sub root ?Y zs* $\wedge$ *sublist (U@V) zs*
$\wedge$ $(\forall as.$ *fwd-sub root ?Y as* $\longrightarrow$ *cost (rev zs) ≤ cost (rev as))*
**using** *no-gap-if-contr-seq-fwd*[*OF insert.prems*(1) *disj fwd nempty fin U V U-def*(4,5)] *zs-def*(1)
**unfolding** *fwd-sub-def unique-set-r-def* **by** *blast*
**then obtain** *xs* **where** *xs-def*:
*fwd-sub root ?Y xs sublist (U@V) xs*
$(\forall as.$ *fwd-sub root ?Y as* $\longrightarrow$ *cost (rev xs) ≤ cost (rev as))*
**by** *blast*

**then have** *cost*: $(\forall as.\ fwd\text{-}sub\ root\ Y\ as \longrightarrow cost\ (rev\ xs) \leq cost\ (rev\ as))$
**using** *zs-def* **by** *fastforce*
**have** *0*: $\forall ys \in (insert\ x\ X).\ \exists U \in Y.\ \exists V \in Y.\ U@V = ys$ **using** *insert.prems*(*7*)
**by** *fastforce*
**then have** $\forall ys \in X.\ \exists U \in Y.\ \exists V \in Y.\ U@V = ys$ **by** *simp*
**then have** $\bigcup (set\ `\ ?Y) = \bigcup (set\ `\ Y)$
**using** *combine-union-sets-set-eq*[*OF insert.prems*(*2*)] **by** *simp*
**then have** $\bigcup (set\ `\ ?X) = \bigcup (set\ `\ ?Y)$
**using** *combine-union-sets-set-eq*[*OF insert.prems*(*2*) *0*] **by** *simp*
**then have** *P-eq*: *unique-set-r root ?X* = *unique-set-r root ?Y* **unfolding** *unique-set-r-def*
**by** *simp*
**have** $\bigwedge ys.\ [\![sublist\ (U@V)\ ys;\ (\forall xs \in ?Y.\ sublist\ xs\ ys)]\!] \Longrightarrow (\forall xs \in ?X.\ sublist$
*xs ys*)
**using** *combine-union-sets-sublists*[*of x*, **where** $Y{=}Y$ **and** $X{=}X$] *U-def*(*3*) **by**
*blast*
**then have** $\bigwedge ys.\ [\![sublist\ (U@V)\ ys;\ fwd\text{-}sub\ root\ ?Y\ ys]\!] \Longrightarrow fwd\text{-}sub\ root\ ?X\ ys$
**unfolding** *P-eq fwd-sub-def* **by** *blast*
**then show** *?case* **using** *xs-def*(*1*,*2*) *cost* **by** *blast*
**qed**

**lemma** *bs-ge-if-geV*:
**assumes** *asi rank r cost*
**and** $\forall xs \in Y.\ \forall ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
**and** $\forall xs \in Y.\ forward\ xs$
**and** $[] \notin Y$
**and** *finite Y*
**and** $U \in Y$
**and** $V \in Y$
**and** *distinct* $(as@U@bs@V@cs)$
**and** $set\ (as@U@bs@V@cs) = \bigcup (set\ `\ Y)$
**and** $\forall xs \in Y.\ sublist\ xs\ (as@U@bs@V@cs)$
**and** $take\ 1\ (as@U@bs@V@cs) = [r]$
**and** $bs \neq []$
**and** $\forall xs \in Y.\ xs \neq U \longrightarrow rank\ (rev\ V) \leq rank\ (rev\ xs)$
**shows** $rank\ (rev\ V) \leq rank\ (rev\ bs)$
**proof** −
**obtain** *as′ bs′ cs′* **where** *bs′-def*: *concat as′* = *as concat bs′* = *bs concat cs′* = *cs*
*set* $(as'@U\#bs'@V\#cs') = Y$
**using** *concat-split-UV*[*OF assms*(*2*,*5*−*10*)] *assms*(*4*,*6*,*7*) **by** *blast*
**have** *tk1*: $take\ 1\ (as@U) = [r]$
**using** *take1-split-nempty*[*of U as*] *assms*(*4*,*6*,*11*) **by** *force*
**have** $\forall z{\in}set\ bs'.\ z \neq U$
**using** *bs′-def*(*2*) *assms*(*4*,*6*,*8*) *concat-all-sublist* **by** (*fastforce dest*!: *empty-if-sublist-dsjnt*)
**then have** *0*: $\forall z{\in}set\ bs'.\ rank\ (rev\ V) \leq rank\ (rev\ z)$
**using** *assms*(*13*) *bs′-def*(*4*) **by** *auto*
**have** $V \neq []$ **using** *assms*(*4*,*7*) **by** *blast*
**have** $[] \notin set\ bs'$ **using** *assms*(*4*) *bs′-def*(*2*,*4*) **by** *auto*
**then show** *?thesis*
**using** *bs-ge-if-all-ge*[*OF assms*(*1*) ‹$V{\neq}[]$›, *of as@U*] *0 bs′-def*(*2*) *tk1 assms*(*3*,*7*,*8*,*12*)

**by** *auto*
**qed**

**lemma** *no-gap-if-geV*:
**assumes** *asi rank root cost*
**and** $\forall\, xs \in Y.\ \forall\, ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
**and** $\forall\, xs \in Y.\ forward\ xs$
**and** $[] \notin Y$
**and** *finite Y*
**and** $U \in Y$
**and** $V \in Y$
**and** *before U V*
**and** $\forall\, xs \in Y.\ xs \neq U \longrightarrow rank\ (rev\ V) \leq rank\ (rev\ xs)$
**and** $\exists\, x.\ fwd\text{-}sub\ root\ Y\ x$
**shows** $\exists\, zs.\ fwd\text{-}sub\ root\ Y\ zs \wedge sublist\ (U@V)\ zs$
$\wedge\ (\forall\, as.\ fwd\text{-}sub\ root\ Y\ as \longrightarrow cost\ (rev\ zs) \leq cost\ (rev\ as))$
**proof** −
**obtain** *zs* **where** *zs-def*:
$set\ zs = \bigcup (set\ `\ Y)$ *distinct zs take 1 zs* = [*root*] *forward zs*
$(\forall\, xs \in Y.\ sublist\ xs\ zs)\ (\forall\, as.\ fwd\text{-}sub\ root\ Y\ as \longrightarrow cost\ (rev\ zs) \leq cost\ (rev\ as))$
**using** *forward-UV-lists-argmin-ex*[*OF assms*(*10*), *of* $\lambda x.\ cost\ (rev\ x)$]
**unfolding** *fwd-sub-def unique-set-r-def* **by** *blast*
**then have** *hd zs* = *root* **using** *hd-eq-take1* **by** *fast*
**then obtain** *as bs cs* **where** *bs-def*: *as* @ *U* @ *bs* @ *V* @ *cs* = *zs*
**using** *sublist-before-if-before zs-def*(*2*,*4*,*5*) *assms*(*6*−*8*) **by** *blast*
**then have** *bs-prems*: *distinct* (*as*@*U*@*bs*@*V*@*cs*) *set* (*as*@*U*@*bs*@*V*@*cs*) = $\bigcup$(*set* ` *Y*)
$\forall\, xs \in Y.\ sublist\ xs\ (as@U@bs@V@cs)\ take\ 1\ (as@U@bs@V@cs) = [root]$
**using** *zs-def*(*1*−*5*) **by** *auto*
**show** *?thesis*
**proof**(*cases bs* = []
**case** *True*
**then have** *sublist* (*U*@*V*) *zs* **using** *bs-def sublist-def* **by** *force*
**then show** *?thesis* **using** *zs-def* **unfolding** *fwd-sub-def unique-set-r-def* **by** *blast*
**next**
**case** *False*
**then have** *rank-le*: *rank* (*rev V*) ≤ *rank* (*rev bs*)
**using** *bs-ge-if-geV*[*OF assms*(*1*−*7*) *bs-prems False assms*(*9*)] **by** *blast*
**have** *0*: *distinct* ((*as*@*U*)@*V*@*bs*@*cs*) **using** *bs-def zs-def*(*2*) **by** *auto*
**have** *take 1* (*as*@*U*) = [*root*]
**using** *bs-def assms*(*4*,*6*) *take1-split-nempty*[*of U as*] *zs-def*(*3*) **by** *fastforce*
**then have** *1*: *take 1* (*as*@*U*@*V*@*bs*@*cs*) = [*root*]
**using** *take1-singleton-app*[*of as*@*U root V*@*bs*@*cs*] **by** *simp*
**have** *2*: $\forall\, xs \in Y.$ *sublist xs* (*as*@*U*@*V*@*bs*@*cs*)
**using** *zs-def*(*5*) *bs-def sublists-preserv-move-VY-all*[*OF assms*(*2*,*6*,*7*)] *assms*(*4*,*6*)
**by** *blast*
**have** $V \neq []$ **using** *assms*(*4*,*7*) **by** *blast*

**have** *cost* (*rev* (*as*@*U*@*V*@*bs*@*cs*)) ≤ *cost* (*rev zs*)
**using** *asi-le-rfst*[*OF assms*(*1*) *rank-le* ‹*V*≠[]› *False 0*] *1 zs-def*(*3*) *bs-def* **by** *simp*
**then have** *cost-le*: ∀ *ys*. *fwd-sub root Y ys* ⟶ *cost* (*rev* (*as*@*U*@*V*@*bs*@*cs*)) ≤ *cost* (*rev ys*)
**using** *zs-def*(*6*) **by** *fastforce*
**have** *forward* (*as*@*U*@*V*@*bs*@*cs*)
**using** *move-mid-backward-if-noarc assms*(*8*) *zs-def*(*4*) *bs-def* **by** *blast*
**moreover have** *set* (*as*@*U*@*V*@*bs*@*cs*) = ⋃(*set* ‘ *Y*) **using** *bs-def zs-def*(*1*) **by** *fastforce*
**ultimately have** *fwd-sub root Y* (*as*@*U*@*V*@*bs*@*cs*)
**unfolding** *fwd-sub-def unique-set-r-def* **using** *0 1 2* **by** *auto*
**moreover have** *sublist* (*U*@*V*) (*as*@*U*@*V*@*bs*@*cs*) **unfolding** *sublist-def* **by** *fastforce*
**ultimately show** *?thesis* **using** *cost-le* **by** *blast*
**qed**
**qed**

**lemma** *app-UV-set-optimal-cost*:
**assumes** *asi rank root cost*
**and** ∀ *xs* ∈ *Y*. ∀ *ys* ∈ *Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}
**and** ∀ *xs* ∈ *Y*. *forward xs*
**and** [] ∉ *Y*
**and** *finite Y*
**and** *U* ∈ *Y*
**and** *V* ∈ *Y*
**and** *before U V*
**and** ∀ *xs* ∈ *Y*. *xs* ≠ *U* ⟶ *rank* (*rev V*) ≤ *rank* (*rev xs*)
**and** ∃ *x*. *fwd-sub root Y x*
**shows** ∃ *zs*. *fwd-sub root* ({*U*@*V*} ∪ {*x*. *x* ∈ *Y* ∧ *x* ≠ *U* ∧ *x* ≠ *V*}) *zs*
∧ (∀ *as*. *fwd-sub root Y as* ⟶ *cost* (*rev zs*) ≤ *cost* (*rev as*))
**proof** −
**have** *P-eq*: *unique-set-r root Y* = *unique-set-r root* ({*U*@*V*} ∪ {*x*. *x* ∈ *Y* ∧ *x* ≠ *U* ∧ *x* ≠ *V*})
**unfolding** *unique-set-r-def* **using** *assms*(*6*,*7*) **by** *auto*
**have** ∃ *zs*. *fwd-sub root Y zs* ∧ *sublist* (*U*@*V*) *zs*
∧ (∀ *as*. *fwd-sub root Y as* ⟶ *cost* (*rev zs*) ≤ *cost* (*rev as*))
**using** *no-gap-if-geV*[*OF assms*(*1*−*10*)] **by** *blast*
**then show** *?thesis* **unfolding** *P-eq fwd-sub-def* **by** *blast*
**qed**

**end**

**context** *tree-query-graph*
**begin**

**lemma** *no-cross-ldeep-rev-if-forward*:
**assumes** *xs* ≠ [] **and** *r* ∈ *verts G* **and** *directed-tree.forward* (*dir-tree-r r*) (*rev xs*)

**shows** *no-cross-products* (*create-ldeep-rev xs*)
**using** *assms* **proof**(*induction xs rule*: *create-ldeep-rev.induct*)
**case** (*3 x y ys*)
**then interpret** *T*: *directed-tree dir-tree-r r r* **using** *directed-tree-r* **by** *blast*
**have** *split*: *create-ldeep-rev* (*x*#*y*#*ys*) = *Join* (*create-ldeep-rev* (*y*#*ys*)) (*Relation x*) **by** *simp*
**have** *rev* (*x*#*y*#*ys*) ! (*length* (*y*#*ys*)) = *x* **using** *nth-append-length*[*of rev* (*y*#*ys*)] **by** *simp*
**moreover have** *length* (*y*#*ys*) ∈ {*1*..*length* (*rev* (*x*#*y*#*ys*)) − *1*} **by** *simp*
**ultimately obtain** *j* **where** *j-def*: *j* < (*length* (*y*#*ys*)) *rev* (*x*#*y*#*ys*)!*j* →*dir-tree-r r x*
**using** *3.prems*(*3*) **unfolding** *T.forward-def* **by** *fastforce*
**then have** *rev* (*x*#*y*#*ys*)!*j* ∈ *set* (*y*#*ys*)
**using** *nth-mem*[*of j rev* (*y*#*ys*)] **by** (*auto simp add*: *nth-append*)
**then have** ∃ *x*′∈*relations* (*create-ldeep-rev* (*y*#*ys*)). *x*′ →*dir-tree-r r x*
**using** *j-def*(*2*) *create-ldeep-rev-relations*[*of y*#*ys*] **by** *blast*
**then have** *1*: ∃ *x*′∈*relations* (*create-ldeep-rev* (*y*#*ys*)). *x*′ →*G x*
**using** *assms*(*2*) *dir-tree-r-dom-in-G* **by** *blast*
**have** *T.forward* (*rev* (*y*#*ys*)) **using** *3.prems*(*3*) *T.forward-cons* **by** *blast*
**then show** *?case* **using** *1 3* **by** *simp*
**qed**(*auto*)

**lemma** *no-cross-ldeep-if-forward*:
⟦*xs* ≠ []; *r* ∈ *verts G*; *directed-tree.forward* (*dir-tree-r r*) *xs*⟧
⟹ *no-cross-products* (*create-ldeep xs*)
**unfolding** *create-ldeep-def* **using** *no-cross-ldeep-rev-if-forward* **by** *simp*

**lemma** *no-cross-ldeep-if-forward′*:
⟦*set xs* = *verts G*; *r* ∈ *verts G*; *directed-tree.forward* (*dir-tree-r r*) *xs*⟧
⟹ *no-cross-products* (*create-ldeep xs*)
**using** *no-cross-ldeep-if-forward*[*of xs*] **by** *fastforce*

**lemma** *forward-if-ldeep-rev-no-cross*:
**assumes** *r* ∈ *verts G* **and** *no-cross-products* (*create-ldeep-rev xs*)
**and** *hd* (*rev xs*) = *r* **and** *distinct xs*
**shows** *directed-tree.forward-arcs* (*dir-tree-r r*) *xs*
**using** *assms* **proof**(*induction xs rule*: *create-ldeep-rev.induct*)
**case** *1*
**then show** *?case* **using** *directed-tree-r directed-tree.forward-arcs.simps*(*1*) **by** *fast*
**next**
**case** (*2 x*)
**then show** *?case* **using** *directed-tree-r directed-tree.forward-arcs.simps*(*2*) **by** *fast*
**next**
**case** (*3 x y ys*)
**then interpret** *T*: *directed-tree dir-tree-r r r* **using** *directed-tree-r* **by** *blast*
**have** *hd* (*rev* (*y* # *ys*)) = *r* **using** *3.prems*(*3*) *hd-append2*[*of rev* (*y*#*ys*) [*x*]] **by** *simp*

**then have** *ind*: *T.forward-arcs* (*y*#*ys*) **using** *3* **by** *fastforce*
**have** *matching*: *matching-rels* (*create-ldeep-rev* (*x*#*y*#*ys*))
**using** *matching-rels-if-no-cross 3.prems*(*2*) **by** *simp*
**have** *r* ∈ *relations* (*create-ldeep-rev* (*x*#*y*#*ys*)) **using** *3.prems*(*3*)
**using** *create-ldeep-rev-relations*[*of x*#*y*#*ys*] *hd-rev*[*of x*#*y*#*ys*] **by** *simp*
**then obtain** *p*′ **where** *p*′*-def*:
*awalk r p*′ *x* ∧ *set* (*awalk-verts r p*′) ⊆ *relations* (*create-ldeep-rev* (*x*#*y*#*ys*))
**using** *no-cross-awalk*[*OF matching 3.prems*(*2*)] **by** *force*
**then obtain** *p* **where** *p-def*:
*apath r p x set* (*awalk-verts r p*) ⊆ *relations* (*create-ldeep-rev* (*x*#*y*#*ys*))
**using** *apath-awalk-to-apath awalk-to-apath-verts-subset* **by** *blast*
**then have** *pre-digraph.apath* (*dir-tree-r r*) *r p x* **using** *apath-in-dir-if-apath-G*
**by** *blast*
**moreover have** *r* ≠ *x*
**using** *3.prems*(*3*,*4*) *T.no-back-arcs.cases*[*of rev* (*x*#*y*#*ys*)] *distinct-first-uneq-last*[*of*
*x*]
**by** *fastforce*
**ultimately obtain** *u* **where** *u-def*:
*u* →*dir-tree-r r x u* ∈ *set* (*pre-digraph.awalk-verts* (*dir-tree-r r*) *r p*)
**using** *p-def*(*2*) *T.awalk-verts-dom-if-uneq T.awalkI-apath* **by** *blast*
**then have** *u* ∈ *relations* (*create-ldeep-rev* (*x*#*y*#*ys*))
**using** *awalk-verts-G-T 3.prems*(*1*) *p-def*(*2*) **by** *auto*
**then have** *u* ∈ *set* (*x*#*y*#*ys*) **by** (*simp add*: *create-ldeep-rev-relations*)
**then show** *?case* **using** *u-def*(*1*) *ind T.forward-arcs.simps*(*3*) *T.loopfree.adj-not-same*
**by** *auto*
**qed**

**lemma** *forward-if-ldeep-no-cross*:
⟦*r* ∈ *verts G*; *no-cross-products* (*create-ldeep xs*); *hd xs* = *r*; *distinct xs*⟧
⟹ *directed-tree.forward* (*dir-tree-r r*) *xs*
**using** *forward-if-ldeep-rev-no-cross directed-tree.forward-arcs-alt directed-tree-r*
**by** (*fastforce simp*: *create-ldeep-def*)

**lemma** *no-cross-ldeep-iff-forward*:
⟦*xs* ≠ []; *r* ∈ *verts G*; *hd xs* = *r*; *distinct xs*⟧
⟹ *no-cross-products* (*create-ldeep xs*) ⟷ *directed-tree.forward* (*dir-tree-r r*)
*xs*
**using** *forward-if-ldeep-no-cross no-cross-ldeep-if-forward* **by** *blast*

**lemma** *no-cross-if-fwd-ldeep*:
⟦*r* ∈ *verts G*; *left-deep t*; *directed-tree.forward* (*dir-tree-r r*) (*inorder t*)⟧
⟹ *no-cross-products t*
**using** *no-cross-ldeep-if-forward*[*OF inorder-nempty*] **by** *fastforce*

**lemma** *forward-if-ldeep-no-cross*′:
⟦*first-node t* ∈ *verts G*; *distinct-relations t*; *left-deep t*; *no-cross-products t*⟧
⟹ *directed-tree.forward* (*dir-tree-r* (*first-node t*)) (*inorder t*)
**using** *forward-if-ldeep-no-cross* **by** (*simp add*: *first-node-eq-hd distinct-relations-def*)

**lemma** *no-cross-iff-forward-ldeep*:
⟦*first-node t* ∈ *verts G*; *distinct-relations t*; *left-deep t*⟧
⟹ *no-cross-products t* ⟷ *directed-tree.forward* (*dir-tree-r* (*first-node t*)) (*inorder t*)
**using** *no-cross-if-fwd-ldeep forward-if-ldeep-no-cross'* **by** *blast*

**lemma** *sublist-before-if-before*:
**assumes** *hd xs* = *r* **and** *no-cross-products* (*create-ldeep xs*) **and** *r* ∈ *verts G* **and** *distinct xs*
**and** *sublist U xs* **and** *sublist V xs* **and** *directed-tree.before* (*dir-tree-r r*) *U V*
**shows** ∃ *as bs cs*. *as* @ *U* @ *bs* @ *V* @ *cs* = *xs*
**using** *directed-tree.sublist-before-if-before*[*OF directed-tree-r*] *forward-if-ldeep-no-cross assms*
**by** *blast*

**lemma** *nocross-UV-lists-subset*:
{*x*. *set x* = *X* ∧ *distinct x* ∧ *take 1 x* = [*r*]
∧ *no-cross-products* (*create-ldeep x*) ∧ (∀ *xs* ∈ *Y*. *sublist xs x*)}
⊆ {*x*. *set x* = *X* ∧ *distinct x*}
**by** *blast*

**lemma** *nocross-UV-lists-finite*:
*finite xs*
⟹ *finite* {*x*. *set x* = *xs* ∧ *distinct x* ∧ *take 1 x* = [*r*]
∧ *no-cross-products* (*create-ldeep x*) ∧ (∀ *xs* ∈ *Y*. *sublist xs x*)}
**using** *distinct-seteq-finite finite-subset*[*OF nocross-UV-lists-subset*] **by** *auto*

**lemma** *nocross-UV-lists-arg-min-ex-aux*:
⟦*finite ys*; *ys* ≠ {};
*ys* = {*x*. *set x* = *xs* ∧ *distinct x* ∧ *take 1 x* = [*r*]
∧ *no-cross-products* (*create-ldeep x*) ∧ (∀ *xs* ∈ *Y*. *sublist xs x*)}⟧
⟹ ∃ *y* ∈ *ys*. ∀ *z* ∈ *ys*. (*f* :: *'a list* ⇒ *real*) *y* ≤ *f z*
**using** *arg-min-if-finite*(*1*)[*of ys f*] *arg-min-least*[*of ys*, **where** *?f* = *f*] **by** *auto*

**lemma** *nocross-UV-lists-arg-min-ex*:
⟦*finite xs*; *ys* ≠ {};
*ys* = {*x*. *set x* = *xs* ∧ *distinct x* ∧ *take 1 x* = [*r*]
∧ *no-cross-products* (*create-ldeep x*) ∧ (∀ *xs* ∈ *Y*. *sublist xs x*)}⟧
⟹ ∃ *y* ∈ *ys*. ∀ *z* ∈ *ys*. (*f* :: *'a list* ⇒ *real*) *y* ≤ *f z*
**using** *nocross-UV-lists-finite nocross-UV-lists-arg-min-ex-aux* **by** *auto*

**lemma** *nocross-UV-lists-argmin-ex*:
**fixes** *f* :: *'a list* ⇒ *real*
**assumes** *P* = (λ*x*. *set x* = *X* ∧ *distinct x* ∧ *take 1 x* = [*r*])
**and** *Q* = (λ*ys*. *P ys* ∧ *no-cross-products* (*create-ldeep ys*) ∧ (∀ *xs* ∈ *Y*. *sublist xs ys*))
**and** ∃ *x*. *Q x*
**shows** ∃ *zs*. *Q zs* ∧ (∀ *as*. *Q as* ⟶ *f zs* ≤ *f as*)
**using** *nocross-UV-lists-arg-min-ex*[*of X* {*x*. *Q x*}] **using** *assms* **by** *fastforce*

**lemma** *no-gap-if-contr-seq*:
**fixes** $Y$ $r$
**defines** $X \equiv \bigcup (set\ `\ Y)$
**defines** $P \equiv (\lambda ys.\ set\ ys = X \wedge distinct\ ys \wedge take\ 1\ ys = [r])$
**defines** $Q \equiv (\lambda ys.\ P\ ys \wedge no\text{-}cross\text{-}products\ (create\text{-}ldeep\ ys) \wedge (\forall xs \in Y.\ sublist\ xs\ ys))$
**assumes** *asi rank r c*
**and** $\forall xs \in Y.\ \forall ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
**and** $\forall xs \in Y.\ directed\text{-}tree.forward\ (dir\text{-}tree\text{-}r\ r)\ xs$
**and** $[] \notin Y$
**and** *finite* $Y$
**and** $U \in Y$
**and** $V \in Y$
**and** $r \in verts\ G$
**and** $directed\text{-}tree.before\ (dir\text{-}tree\text{-}r\ r)\ U\ V$
**and** $rank\ (rev\ V) \leq rank\ (rev\ U)$
**and** $\bigwedge xs.\ [\![xs \in Y;\ \exists y{\in}set\ xs.\ \neg(\exists x'{\in}set\ V.\ x' \rightarrow^+_{dir\text{-}tree\text{-}r\ r} y) \wedge (\exists x{\in}set\ U.\ x \rightarrow^+_{dir\text{-}tree\text{-}r\ r} y);\ xs \neq U]\!] \Longrightarrow rank\ (rev\ V) \leq rank\ (rev\ xs)$
**and** $\exists x.\ Q\ x$
**shows** $\exists zs.\ Q\ zs \wedge sublist\ (U@V)\ zs \wedge (\forall as.\ Q\ as \longrightarrow c\ (rev\ zs) \leq c\ (rev\ as))$
**proof** −
**interpret** $T$: *directed-tree dir-tree-r r r* **using** *assms(11) directed-tree-r* **by** *auto*
**let** *?Q* $= (\lambda ys.\ P\ ys \wedge T.forward\ ys \wedge (\forall xs \in Y.\ sublist\ xs\ ys))$
**have** *?Q* $= Q$
**using** *no-cross-ldeep-iff-forward assms(11,2,3) hd-eq-take1 nempty-if-take1* [**where** *r=r*] **by** *fast*
**then show** *?thesis*
**using** *T.no-gap-if-contr-seq-fwd*[*OF assms(4−10,12−14)*] *assms(15,1,2)*
**unfolding** *T.fwd-sub-def unique-set-r-def* **by** *auto*
**qed**

**end**

## 10.3 Arc Invariants

**function** *path-lverts* :: $('a\ list, 'b)\ dtree \Rightarrow 'a \Rightarrow 'a\ set$ **where**
*path-lverts* $(Node\ r\ \{|(t,e)|\})\ x = (if\ x \in set\ r\ then\ \{\}\ else\ set\ r \cup path\text{-}lverts\ t\ x)$
$|\ \forall x.\ xs \neq \{|x|\} \Longrightarrow path\text{-}lverts\ (Node\ r\ xs)\ x = (if\ x \in set\ r\ then\ \{\}\ else\ set\ r)$
**by** (*metis darcs-mset.cases old.prod.exhaust*) *fast+*
**termination by** *lexicographic-order*

**definition** *path-lverts-list* :: $('a\ list \times 'b)\ list \Rightarrow 'a \Rightarrow 'a\ set$ **where**
*path-lverts-list xs x* $= (\bigcup (t,e){\in}\ set\ (takeWhile\ (\lambda(t,e).\ x \notin set\ t)\ xs).\ set\ t)$

**definition** *dom-children* :: $('a\ list, 'b)\ dtree \Rightarrow ('a, 'b)\ pre\text{-}digraph \Rightarrow bool$ **where**
*dom-children t1 T* $= (\forall t \in fst\ `\ fset\ (sucs\ t1).\ \forall x \in dverts\ t.$

$\exists\, r \in set\ (root\ t1) \cup path\text{-}lverts\ t\ (hd\ x).\ r \rightarrow_T hd\ x)$

**abbreviation** *children-deg1* :: $(('a,'b)\ dtree \times 'b)\ fset \Rightarrow (('a,'b)\ dtree \times 'b)\ set$
**where**
*children-deg1 xs* $\equiv \{(t,e).\ (t,e) \in fset\ xs \wedge max\text{-}deg\ t \leq 1\}$

**lemma** *path-lverts-subset-dlverts*: *path-lverts t x* $\subseteq$ *dlverts t*
**by**(*induction t x rule*: *path-lverts.induct*) *auto*

**lemma** *path-lverts-to-list-eq*:
*path-lverts t x* = *path-lverts-list* (*dtree-to-list* (*Node r0* $\{|(t,e)|\}$)) *x*
**by** (*induction t rule*: *dtree-to-list.induct*) (*auto simp*: *path-lverts-list-def*)

**lemma** *path-lverts-from-list-eq*:
*path-lverts* (*dtree-from-list r0 ys*) *x* = *path-lverts-list* ((*r0*,*e0*)#*ys*) *x*
**unfolding** *path-lverts-list-def* **using** *path-lverts.simps*(*2*)[*of* $\{||\}$]
**by** (*induction ys rule*: *dtree-from-list.induct*) (*force*, *cases x* $\in$ *set r0*, *auto*)

**lemma** *path-lverts-child-union-root-sub*:
**assumes** *t2* $\in$ *fst* ' *fset* (*sucs t1*)
**shows** *path-lverts t1 x* $\subseteq$ *set* (*root t1*) $\cup$ *path-lverts t2 x*
**proof**(*cases* $\forall$ *x. sucs t1* $\neq \{|x|\}$)
**case** *True*
**then show** *?thesis* **using** *path-lverts.simps*(*2*)[*of sucs t1 root t1*] **by** *simp*
**next**
**case** *False*
**then obtain** *e2* **where** *sucs t1* = $\{|(t2,e2)|\}$ **using** *assms* **by** *fastforce*
**then show** *?thesis*
**using** *path-lverts.simps*(*1*)[*of root t1 t2 e2*] *dtree.collapse*[*of t1*]
**by**(*cases x* $\in$ *set* (*root t1*)) *fastforce*+
**qed**

**lemma** *path-lverts-simps1-sucs*:
$[\![x \notin set\ (root\ t1);\ sucs\ t1 = \{|(t2,e2)|\}]\!]$
$\Longrightarrow$ *set* (*root t1*) $\cup$ *path-lverts t2 x* = *path-lverts t1 x*
**using** *path-lverts.simps*(*1*)[*of root t1 t2 e2 x*] *dtree.exhaust-sel*[*of t1*] **by** *argo*

**lemma** *subtree-path-lverts-sub*:
$[\![$*wf-dlverts t1*; *max-deg t1* $\leq$ *1*; *is-subtree* (*Node r xs*) *t1*; *t2* $\in$ *fst* ' *fset xs*; *x*$\in$*set* (*root t2*)$]\!]$
$\Longrightarrow$ *set r* $\subseteq$ *path-lverts t1 x*
**proof**(*induction t1*)
**case** (*Node r1 xs1*)
**then have** *xs1* $\neq \{||\}$ **by** *force*
**then have** *max-deg* (*Node r1 xs1*) = *1*
**using** *Node.prems*(*2*) *empty-if-mdeg-0*[*of r1 xs1*] **by** *fastforce*
**then obtain** *t e* **where** *t-def*: *xs1* = $\{|(t,e)|\}$ **using** *mdeg-1-singleton* **by** *fastforce*
**have** *x-t2*: *x* $\in$ *dlverts t2* **using** *Node.prems*(*5*) *lverts-if-in-verts dtree.set-sel*(*1*)
**by** *fast*

**show** *?case*
**proof**(*cases Node r1 xs1 = Node r xs*)
**case** *True*
**then show** *?thesis* **using** *Node.prems*(*1*,*4*) *x-t2 t-def* **by** *force*
**next**
**case** *False*
**then have** *0*: *is-subtree* (*Node r xs*) *t* **using** *t-def Node.prems*(*3*) **by** *force*
**moreover have** *max-deg t* ≤ *1* **using** *t-def Node.prems*(*2*) *mdeg-ge-child*[*of t e xs1*] **by** *simp*
**moreover have** *x* ∉ *set r1* **using** *t-def x-t2 Node.prems*(*1*,*4*) *0 subtree-in-dlverts* **by** *force*
**ultimately show** *?thesis* **using** *Node.IH t-def Node.prems*(*1*,*4*,*5*) **by** *auto*
**qed**
**qed**

**lemma** *path-lverts-empty-if-roothd*:
**assumes** *root t* ≠ []
**shows** *path-lverts t* (*hd* (*root t*)) = {}
**proof**(*cases* ∀ *x*. *sucs t* ≠ {|*x*|})
**case** *True*
**then show** *?thesis* **using** *path-lverts.simps*(*2*)[*of sucs t root t*] **by** *force*
**next**
**case** *False*
**then obtain** *t1 e1* **where** *t1-def*: *sucs t* = {|(*t1*, *e1*)|} **by** *auto*
**then have** *path-lverts t* (*hd* (*root t*)) =
(*if hd* (*root t*) ∈ *set* (*root t*) *then* {} *else set* (*root t*) ∪ *path-lverts t1* (*hd* (*root t*)))
**using** *path-lverts.simps*(*1*) *dtree.collapse* **by** *metis*
**then show** *?thesis* **using** *assms* **by** *simp*
**qed**

**lemma** *path-lverts-subset-root-if-childhd*:
**assumes** *t1* ∈ *fst* ‘ *fset* (*sucs t*) **and** *root t1* ≠ []
**shows** *path-lverts t* (*hd* (*root t1*)) ⊆ *set* (*root t*)
**proof**(*cases* ∀ *x*. *sucs t* ≠ {|*x*|})
**case** *True*
**then show** *?thesis* **using** *path-lverts.simps*(*2*)[*of sucs t root t*] **by** *simp*
**next**
**case** *False*
**then obtain** *e1* **where** *sucs t* = {|(*t1*, *e1*)|} **using** *assms*(*1*) **by** *fastforce*
**then have** *path-lverts t* (*hd* (*root t1*)) =
(*if hd* (*root t1*) ∈ *set* (*root t*) *then* {} *else set* (*root t*) ∪ *path-lverts t1* (*hd* (*root t1*)))
**using** *path-lverts.simps*(*1*) *dtree.collapse* **by** *metis*
**then show** *?thesis* **using** *path-lverts-empty-if-roothd*[*OF assms*(*2*)] **by** *auto*
**qed**

**lemma** *path-lverts-list-merge-supset-xs-notin*:
∀ *v* ∈ *fst* ‘ *set ys*. *a* ∉ *set v*

$\Longrightarrow$ *path-lverts-list xs a* $\subseteq$ *path-lverts-list* (*Sorting-Algorithms.merge cmp xs ys*) *a*
**proof**(*induction xs ys taking*: *cmp rule*: *Sorting-Algorithms.merge.induct*)
**case** (*3 x xs y ys*)
**obtain** *v1 e1* **where** *v1-def*[*simp*]: *x* = (*v1*,*e1*) **by** *force*
**obtain** *v2 e2* **where** *y* = (*v2*,*e2*) **by** *force*
**then show** *?case* **using** *3* **by** (*auto simp*: *path-lverts-list-def*)
**qed** (*auto simp*: *path-lverts-list-def*)

**lemma** *path-lverts-list-merge-supset-ys-notin*:
$\forall$ *v* $\in$ *fst* ‘ *set xs*. *a* $\notin$ *set v*
$\Longrightarrow$ *path-lverts-list ys a* $\subseteq$ *path-lverts-list* (*Sorting-Algorithms.merge cmp xs ys*) *a*
**proof**(*induction xs ys taking*: *cmp rule*: *Sorting-Algorithms.merge.induct*)
**case** (*3 x xs y ys*)
**obtain** *v1 e1* **where** *v1-def*[*simp*]: *x* = (*v1*,*e1*) **by** *force*
**obtain** *v2 e2* **where** *y* = (*v2*,*e2*) **by** *force*
**then show** *?case* **using** *3* **by** (*auto simp*: *path-lverts-list-def*)
**qed** (*auto simp*: *path-lverts-list-def*)

**lemma** *path-lverts-list-merge-supset-xs*:
⟦$\exists$ *v* $\in$ *fst* ‘ *set xs*. *a* $\in$ *set v*; $\forall$ *v1* $\in$ *fst* ‘ *set xs*. $\forall$ *v2* $\in$ *fst* ‘ *set ys*. *set v1* $\cap$ *set v2* = {}⟧
$\Longrightarrow$ *path-lverts-list xs a* $\subseteq$ *path-lverts-list* (*Sorting-Algorithms.merge cmp xs ys*) *a*
**using** *path-lverts-list-merge-supset-xs-notin* **by** *fast*

**lemma** *path-lverts-list-merge-supset-ys*:
⟦$\exists$ *v* $\in$ *fst* ‘ *set ys*. *a* $\in$ *set v*; $\forall$ *v1* $\in$ *fst* ‘ *set xs*. $\forall$ *v2* $\in$ *fst* ‘ *set ys*. *set v1* $\cap$ *set v2* = {}⟧
$\Longrightarrow$ *path-lverts-list ys a* $\subseteq$ *path-lverts-list* (*Sorting-Algorithms.merge cmp xs ys*) *a*
**using** *path-lverts-list-merge-supset-ys-notin* **by** *fast*

**lemma** *dom-children-if-all-singletons*:
$\forall$ (*t1*,*e1*) $\in$ *fset xs*. *dom-children* (*Node r* {|(*t1*, *e1*)|}) *T* $\Longrightarrow$ *dom-children* (*Node r xs*) *T*
**by** (*auto simp*: *dom-children-def*)

**lemma** *dom-children-all-singletons*:
⟦*dom-children* (*Node r xs*) *T*; (*t1*,*e1*) $\in$ *fset xs*⟧ $\Longrightarrow$ *dom-children* (*Node r* {|(*t1*, *e1*)|}) *T*
**by** (*auto simp*: *dom-children-def*)

**lemma** *dom-children-all-singletons'*:
⟦*dom-children* (*Node r xs*) *T*; *t1*$\in$ *fst* ‘ *fset xs*⟧ $\Longrightarrow$ *dom-children* (*Node r* {|(*t1*, *e1*)|}) *T*
**by** (*auto simp*: *dom-children-def*)

**lemma** *root-arc-if-dom-root-child-nempty*:
⟦*dom-children* (*Node r xs*) *T*; *t1* ∈ *fst* ‘ *fset xs*; *root t1* ≠ []⟧
⟹ ∃ *x*∈*set r*. ∃ *y*∈*set* (*root t1*). *x* →$_T$ *y*
**unfolding** *dom-children-def* **using** *dtree.set-sel*(*1*) *path-lverts-empty-if-roothd*[*of t1*]
**by** *fastforce*

**lemma** *root-arc-if-dom-root-child-wfdlverts*:
⟦*dom-children* (*Node r xs*) *T*; *t1* ∈ *fst* ‘ *fset xs*; *wf-dlverts t1*⟧
⟹ ∃ *x*∈*set r*. ∃ *y*∈*set* (*root t1*). *x* →$_T$ *y*
**using** *root-arc-if-dom-root-child-nempty dtree.set-sel*(*1*)[*of t1*] *empty-notin-wf-dlverts*
**by** *fastforce*

**lemma** *root-arc-if-dom-wfdlverts*:
⟦*dom-children* (*Node r xs*) *T*; *t1* ∈ *fst* ‘ *fset xs*; *wf-dlverts* (*Node r xs*)⟧
⟹ ∃ *x*∈*set r*. ∃ *y*∈*set* (*root t1*). *x* →$_T$ *y*
**using** *root-arc-if-dom-root-child-wfdlverts*[*of r xs T t1*] **by** *fastforce*

**lemma** *children-deg1-sub-xs*: {(*t*,*e*). (*t*,*e*) ∈ *fset xs* ∧ *max-deg t* ≤ *1*} ⊆ (*fset xs*)
**by** *blast*

**lemma** *finite-children-deg1*: *finite* {(*t*,*e*). (*t*,*e*) ∈ *fset xs* ∧ *max-deg t* ≤ *1*}
**using** *children-deg1-sub-xs*[*of xs*] **by** (*simp add*: *finite-subset*)

**lemma** *finite-children-deg1′*: {(*t*,*e*). (*t*,*e*) ∈ *fset xs* ∧ *max-deg t* ≤ *1*} ∈ {*A*. *finite A*}
**using** *finite-children-deg1* **by** *blast*

**lemma** *children-deg1-fset-id*[*simp*]: *fset* (*Abs-fset* (*children-deg1 xs*)) = *children-deg1 xs*
**using** *Abs-fset-inverse*[*OF finite-children-deg1′*] **by** *auto*

**lemma** *xs-sub-children-deg1*: ∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1* ⟹ (*fset xs*) ⊆ *children-deg1 xs*
**by** *auto*

**lemma** *children-deg1-full*:
∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1* ⟹ (*Abs-fset* (*children-deg1 xs*)) = *xs*
**using** *xs-sub-children-deg1*[*of xs*] *children-deg1-sub-xs*[*of xs*] **by** (*simp add*: *fset-inverse*)

**locale** *ranked-dtree-with-orig* = *ranked-dtree t rank cmp* + *directed-tree T root*
**for** *t* :: (′*a list*, ′*b*) *dtree* **and** *rank cost cmp* **and** *T* :: (′*a*, ′*b*) *pre-digraph* **and** *root* +
**assumes** *asi-rank*: *asi rank root cost*
**and** *dom-mdeg-gt1*:
⟦*is-subtree* (*Node r xs*) *t*; *t1* ∈ *fst* ‘ *fset xs*; *max-deg* (*Node r xs*) > *1*⟧
⟹ ∃ *v* ∈ *set r*. *v* →$_T$ *hd* (*Dtree.root t1*)
**and** *dom-sub-contr*:
⟦*is-subtree* (*Node r xs*) *t*; *t1* ∈ *fst* ‘ *fset xs*;

$\exists\, v\ t2\ e2.\ is\text{-}subtree\ (Node\ v\ \{|(t2,e2)|\})\ (Node\ r\ xs) \wedge rank\ (rev\ (Dtree.root\ t2)) < rank\ (rev\ v)\rrbracket$
$\Longrightarrow \exists\, v \in set\ r.\ v \rightarrow_T hd\ (Dtree.root\ t1)$
**and** *dom-contr*:
$\llbracket is\text{-}subtree\ (Node\ r\ \{|(t1,e1)|\})\ t;\ rank\ (rev\ (Dtree.root\ t1)) < rank\ (rev\ r);$
$max\text{-}deg\ (Node\ r\ \{|(t1,e1)|\}) = 1\rrbracket$
$\Longrightarrow dom\text{-}children\ (Node\ r\ \{|(t1,e1)|\})\ T$
**and** *dom-wedge*:
$\llbracket is\text{-}subtree\ (Node\ r\ xs)\ t;\ fcard\ xs > 1\rrbracket$
$\Longrightarrow dom\text{-}children\ (Node\ r\ (Abs\text{-}fset\ (children\text{-}deg1\ xs)))\ T$
**and** *arc-in-dlverts*:
$\llbracket is\text{-}subtree\ (Node\ r\ xs)\ t;\ x \in set\ r;\ x \rightarrow_T y\rrbracket \Longrightarrow y \in dlverts\ (Node\ r\ xs)$
**and** *verts-conform*: $v \in dverts\ t \Longrightarrow seq\text{-}conform\ v$
**and** *verts-distinct*: $v \in dverts\ t \Longrightarrow distinct\ v$
**begin**

**lemma** *dom-contr′*:
$\llbracket is\text{-}subtree\ (Node\ r\ \{|(t1,e1)|\})\ t;\ rank\ (rev\ (Dtree.root\ t1)) < rank\ (rev\ r);$
$max\text{-}deg\ (Node\ r\ \{|(t1,e1)|\}) \leq 1\rrbracket$
$\Longrightarrow dom\text{-}children\ (Node\ r\ \{|(t1,e1)|\})\ T$
**using** *dom-contr mdeg-ge-sub mdeg-singleton*[*of r t1*] **by** (*simp add*: *fcard-single-1*)

**lemma** *dom-self-contr*:
$\llbracket is\text{-}subtree\ (Node\ r\ \{|(t1,e1)|\})\ t;\ rank\ (rev\ (Dtree.root\ t1)) < rank\ (rev\ r)\rrbracket$
$\Longrightarrow \exists\, v \in set\ r.\ v \rightarrow_T hd\ (Dtree.root\ t1)$
**using** *dom-sub-contr* **by** *fastforce*

**lemma** *dom-wedge-full*:
$\llbracket is\text{-}subtree\ (Node\ r\ xs)\ t;\ fcard\ xs > 1;\ \forall\, t \in fst\ `\ fset\ xs.\ max\text{-}deg\ t \leq 1\rrbracket$
$\Longrightarrow dom\text{-}children\ (Node\ r\ xs)\ T$
**using** *dom-wedge children-deg1-full* **by** *fastforce*

**lemma** *dom-wedge-singleton*:
$\llbracket is\text{-}subtree\ (Node\ r\ xs)\ t;\ fcard\ xs > 1;\ t1 \in fst\ `\ fset\ xs;\ max\text{-}deg\ t1 \leq 1\rrbracket$
$\Longrightarrow dom\text{-}children\ (Node\ r\ \{|(t1,e1)|\})\ T$
**using** *dom-children-all-singletons′ dom-wedge children-deg1-fset-id* **by** *fastforce*

**lemma** *arc-to-dverts-in-subtree*:
$\llbracket is\text{-}subtree\ (Node\ r\ xs)\ t;\ x \in set\ r;\ x \rightarrow_T y;\ y \in set\ v;\ v \in dverts\ t\rrbracket$
$\Longrightarrow v \in dverts\ (Node\ r\ xs)$
**using** *list-in-verts-if-lverts*[*OF arc-in-dlverts*] *dverts-same-if-set-wf*[*OF wf-lverts*]
*dverts-subtree-subset* **by** *blast*

**lemma** *dlverts-arc-in-dlverts*:
$\llbracket is\text{-}subtree\ t1\ t;\ x \rightarrow_T y;\ x \in dlverts\ t1\rrbracket \Longrightarrow y \in dlverts\ t1$
**proof**(*induction t1*)
**case** (*Node r xs*)
**then show** *?case*
**proof**(*cases* $x \in set\ r$)

```
    case True
    then show ?thesis using arc-in-dlverts Node.prems(1,2) by blast
  next
    case False
    then obtain t2 e2 where t2-def: (t2,e2) ∈ fset xs x ∈ dlverts t2
      using Node.prems(3) by auto
    then have is-subtree t2 (Node r xs) using subtree-if-child
      by (metis image-iff prod.sel(1))
    then have is-subtree t2 t using Node.prems(1) subtree-trans by blast
    then show ?thesis using Node.IH Node.prems(2) t2-def by fastforce
  qed
qed

lemma dverts-arc-in-dlverts:
  ⟦is-subtree t1 t; v1 ∈ dverts t1; x ∈ set v1; x →T y⟧ ⟹ y ∈ dlverts t1
  using dlverts-arc-in-dlverts by (simp add: lverts-if-in-verts)

lemma dverts-arc-in-dverts:
  assumes is-subtree t1 t
      and v1 ∈ dverts t1
      and x ∈ set v1
      and x →T y
      and y ∈ set v2
      and v2 ∈ dverts t
    shows v2 ∈ dverts t1
proof −
  have x ∈ dlverts t1 using assms(2,3) lverts-if-in-verts by fast
  then obtain v where v-def: v∈dverts t1 y ∈ set v
   using list-in-verts-if-lverts[OF dlverts-arc-in-dlverts] assms(1−4) lverts-if-in-verts
by blast
  then show ?thesis
     using dverts-same-if-set-wf[OF wf-lverts] assms(1,5,6) dverts-subtree-subset
by blast
qed

lemma dlverts-reach1-in-dlverts:
  ⟦x →+T y; is-subtree t1 t; x ∈ dlverts t1⟧ ⟹ y ∈ dlverts t1
  by(induction x y rule: trancl.induct) (auto simp: dlverts-arc-in-dlverts)

lemma dlverts-reach-in-dlverts:
  ⟦x →*T y; is-subtree t1 t; x ∈ dlverts t1⟧ ⟹ y ∈ dlverts t1
  using dlverts-reach1-in-dlverts by blast

lemma dverts-reach1-in-dlverts:
  ⟦is-subtree t1 t; v1 ∈ dverts t1; x ∈ set v1; x →+T y⟧ ⟹ y ∈ dlverts t1
  using dlverts-reach1-in-dlverts by (simp add: lverts-if-in-verts)

lemma dverts-reach-in-dlverts:
  ⟦is-subtree t1 t; v1 ∈ dverts t1; x ∈ set v1; x →*T y⟧ ⟹ y ∈ dlverts t1
```

**using** *list-in-verts-iff-lverts dverts-reach1-in-dlverts* **by** (*cases x=y,fastforce,blast*)

**lemma** *dverts-reach1-in-dverts*:
⟦*is-subtree t1 t*; *v1* ∈ *dverts t1*; *x* ∈ *set v1*; *x* →⁺$_T$ *y*; *y* ∈ *set v2*; *v2* ∈ *dverts t*⟧
⟹ *v2* ∈ *dverts t1*
**by** (*meson dverts-reach1-in-dlverts dverts-arc-in-dverts list-in-verts-if-lverts tranclE*)

**lemma** *dverts-same-if-set-subtree*:
⟦*is-subtree t1 t*; *v1* ∈ *dverts t1*; *x* ∈ *set v1*; *x* ∈ *set v2*; *v2* ∈ *dverts t*⟧ ⟹ *v1* = *v2*
**using** *dverts-same-if-set-wf*[*OF wf-lverts*] *dverts-subtree-subset* **by** *blast*

**lemma** *dverts-reach-in-dverts*:
⟦*is-subtree t1 t*; *v1* ∈ *dverts t1*; *x* ∈ *set v1*; *x* →*$_T$ *y*; *y* ∈ *set v2*; *v2* ∈ *dverts t*⟧
⟹ *v2* ∈ *dverts t1*
**using** *dverts-same-if-set-subtree dverts-reach1-in-dverts* **by** *blast*

**lemma** *dverts-reach1-in-dverts-root*:
⟦*is-subtree t1 t*; *v* ∈ *dverts t*; ∃ *x*∈*set* (*Dtree.root t1*). ∃ *y*∈*set v*. *x* →⁺$_T$ *y*⟧
⟹ *v* ∈ *dverts t1*
**using** *dverts-reach1-in-dverts dtree.set-sel*(*1*) **by** *blast*

**lemma** *dverts-reach1-in-dverts-r*:
⟦*is-subtree* (*Node r xs*) *t*; *v* ∈ *dverts t*; ∃ *x*∈*set r*. ∃ *y*∈*set v*. *x* →⁺$_T$ *y*⟧
⟹ *v* ∈ *dverts* (*Node r xs*)
**using** *dverts-reach1-in-dverts*[*of Node r xs*] **by** (*auto intro*: *dtree.set-intros*(*1*))

**lemma** *dom-mdeg-gt1-subtree*:
⟦*is-subtree tn t*; *is-subtree* (*Node r xs*) *tn*; *t1* ∈ *fst* ‘ *fset xs*; *max-deg* (*Node r xs*) > *1*⟧
⟹ ∃ *v* ∈ *set r*. *v* →$_T$ *hd* (*Dtree.root t1*)
**using** *dom-mdeg-gt1 subtree-trans* **by** *blast*

**lemma** *dom-sub-contr-subtree*:
⟦*is-subtree tn t*; *is-subtree* (*Node r xs*) *tn*; *t1* ∈ *fst* ‘ *fset xs*;
∃ *v t2 e2*. *is-subtree* (*Node v* {|(*t2*,*e2*)|}) (*Node r xs*) ∧ *rank* (*rev* (*Dtree.root t2*)) < *rank* (*rev v*)⟧
⟹ ∃ *v* ∈ *set r*. *v* →$_T$ *hd* (*Dtree.root t1*)
**using** *dom-sub-contr subtree-trans* **by** *blast*

**lemma** *dom-contr-subtree*:
⟦*is-subtree tn t*; *is-subtree* (*Node r* {|(*t1*,*e1*)|}) *tn*; *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*);
*max-deg* (*Node r* {|(*t1*,*e1*)|}) = *1*⟧
⟹ *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T*
**using** *dom-contr subtree-trans* **by** *blast*

**lemma** *dom-wedge-subtree*:

⟦*is-subtree tn t*; *is-subtree* (*Node r xs*) *tn*; *fcard xs* > *1*⟧
⟹ *dom-children* (*Node r* (*Abs-fset* (*children-deg1 xs*))) *T*
**using** *dom-wedge subtree-trans* **by** *blast*

**corollary** *dom-wedge-subtree′*:
*is-subtree tn t* ⟹∀ *r xs*. *is-subtree* (*Node r xs*) *tn* ⟶ *fcard xs* > *1*
⟶ *dom-children* (*Node r* (*Abs-fset* {(*t*, *e*). (*t*, *e*) ∈ *fset xs* ∧ *max-deg t* ≤ *Suc 0*})) *T*
**by** (*auto simp only*: *dom-wedge-subtree One-nat-def*[*symmetric*])

**lemma** *dom-wedge-full-subtree*:
⟦*is-subtree tn t*; *is-subtree* (*Node r xs*) *tn*; *fcard xs* > *1*; ∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*⟧
⟹ *dom-children* (*Node r xs*) *T*
**using** *dom-wedge-full subtree-trans* **by** *fast*

**lemma** *arc-in-dlverts-subtree*:
⟦*is-subtree tn t*; *is-subtree* (*Node r xs*) *tn*; *x* ∈ *set r*; *x* →$_T$ *y*⟧ ⟹ *y* ∈ *dlverts* (*Node r xs*)
**using** *arc-in-dlverts subtree-trans* **by** *blast*

**corollary** *arc-in-dlverts-subtree′*:
*is-subtree tn t* ⟹ ∀ *r xs*. *is-subtree* (*Node r xs*) *tn* ⟶ (∀ *x*. *x* ∈ *set r*
⟶ (∀ *y*. *x* →$_T$ *y* ⟶ *y* ∈ *set r* ∨ (∃ *c*∈*fset xs*. *y* ∈ *dlverts* (*fst c*))))
**using** *arc-in-dlverts-subtree* **by** *simp*

**lemma** *verts-conform-subtree*: ⟦*is-subtree tn t*; *v* ∈ *dverts tn*⟧ ⟹ *seq-conform v*
**using** *verts-conform dverts-subtree-subset* **by** *blast*

**lemma** *verts-distinct-subtree*: ⟦*is-subtree tn t*; *v* ∈ *dverts tn*⟧ ⟹ *distinct v*
**using** *verts-distinct dverts-subtree-subset* **by** *blast*

**lemma** *ranked-dtree-orig-subtree*: *is-subtree x t* ⟹ *ranked-dtree-with-orig x rank cost cmp T root*
**unfolding** *ranked-dtree-with-orig-def ranked-dtree-with-orig-axioms-def*
**by** (*simp add*: *ranked-dtree-subtree directed-tree-axioms dom-mdeg-gt1-subtree dom-contr-subtree*
*dom-sub-contr-subtree dom-wedge-subtree′ arc-in-dlverts-subtree′*
*verts-conform-subtree verts-distinct-subtree asi-rank*)

**corollary** *ranked-dtree-orig-rec*:
⟦*Node r xs* = *t*; (*x*,*e*) ∈ *fset xs*⟧ ⟹ *ranked-dtree-with-orig x rank cost cmp T root*
**using** *ranked-dtree-orig-subtree*[*of x*] *subtree-if-child*[*of x xs*] **by** *force*

**lemma** *child-disjoint-root*:
⟦*is-subtree* (*Node r xs*) *t*; *t1* ∈ *fst* ‘ *fset xs*⟧ ⟹ *set r* ∩ *set* (*Dtree.root t1*) = {}
**using** *wf-dlverts-subtree*[*OF wf-lverts*] *dlverts-eq-dverts-union dtree.set-sel*(*1*) **by** *fastforce*

**lemma** *distint-verts-subtree*:
**assumes** *is-subtree* (*Node r xs*) *t* **and** *t1* ∈ *fst* ‘ *fset xs*
**shows** *distinct* (*r* @ *Dtree.root t1*)
**proof** −
**have** (*Dtree.root t1*) ∈ *dverts t* **using** *dtree.set-sel*(*1*) *assms dverts-subtree-subset*
**by** *fastforce*
**then show** *?thesis*
**using** *verts-distinct assms*(*1*) *dverts-subtree-subset child-disjoint-root*[*OF assms*]
**by** *force*
**qed**

**corollary** *distint-verts-singleton-subtree*:
*is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t* ⟹ *distinct* (*r* @ *Dtree.root t1*)
**using** *distint-verts-subtree* **by** *simp*

**lemma** *dom-between-child-roots*:
**assumes** *is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t* **and** *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*)
**shows** ∃ *x*∈*set r*. ∃ *y*∈*set* (*Dtree.root t1*). *x* →$_T$ *y*
**using** *dom-self-contr*[*OF assms*] *wf-dlverts-subtree*[*OF wf-lverts assms*(*1*)]
*hd-in-set*[*of Dtree.root t1*] *dtree.set-sel*(*1*)[*of t1*] *empty-notin-wf-dlverts*[*of t1*]
**by** *fastforce*

**lemma** *contr-before*:
**assumes** *is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t* **and** *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*)
**shows** *before r* (*Dtree.root t1*)
**proof** −
**have** (*Dtree.root t1*) ∈ *dverts t* **using** *dtree.set-sel*(*1*) *assms*(*1*) *dverts-subtree-subset*
**by** *fastforce*
**then have** *seq-conform* (*Dtree.root t1*) **using** *verts-conform* **by** *simp*
**moreover have** *seq-conform r* **using** *verts-conform assms*(*1*) *dverts-subtree-subset*
**by** *force*
**ultimately show** *?thesis*
**using** *before-def dom-between-child-roots*[*OF assms*] *child-disjoint-root*[*OF assms*(*1*)]
**by** *auto*
**qed**

**lemma** *contr-forward*:
**assumes** *is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t* **and** *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*)
**shows** *forward* (*r*@*Dtree.root t1*)
**proof** −
**have** (*Dtree.root t1*) ∈ *dverts t* **using** *dtree.set-sel*(*1*) *assms*(*1*) *dverts-subtree-subset*
**by** *fastforce*
**then have** *seq-conform* (*Dtree.root t1*) **using** *verts-conform* **by** *simp*
**moreover have** *seq-conform r* **using** *verts-conform assms*(*1*) *dverts-subtree-subset*
**by** *force*
**ultimately show** *?thesis*

**using** *seq-conform-def forward-arcs-alt dom-self-contr assms forward-app* **by** *simp*
**qed**

**lemma** *contr-seq-conform*:
⟦*is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t*; *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*)⟧
⟹ *seq-conform* (*r* @ *Dtree.root t1*)
**using** *seq-conform-if-before contr-before* **by** *simp*

**lemma** *verts-forward*: ∀ *v* ∈ *dverts t*. *forward v*
**using** *seq-conform-alt verts-conform* **by** *simp*

**lemma** *dverts-reachable1-if-dom-children-aux-root*:
**assumes** ∀ *v*∈*dverts* (*Node r xs*). ∃ *x*∈*set r0* ∪ *X* ∪ *path-lverts* (*Node r xs*) (*hd v*). *x* →$_T$ *hd v*
**and** ∀ *y*∈*X*. ∃ *x*∈*set r0*. *x* →$^+{}_T$ *y*
**and** *forward r*
**shows** ∀ *y*∈*set r*. ∃ *x*∈*set r0*. *x* →$^+{}_T$ *y*
**proof**(*cases r* = [])
**case** *False*
**then have** *path-lverts* (*Node r xs*) (*hd r*) = {}
**using** *path-lverts-empty-if-roothd*[*of Node r xs*] **by** *simp*
**then obtain** *x* **where** *x-def*: *x*∈*set r0* ∪ *X x* →$_T$ *hd r* **using** *assms*(*1*) **by** *auto*
**then have** *hd r* ∈ *verts T* **using** *adj-in-verts*(*2*) **by** *auto*
**then have** ∀ *y*∈*set r*. *x* →$^+{}_T$ *y*
**using** *hd-reach-all-forward x-def*(*2*) *assms*(*3*) *reachable1-reachable-trans* **by** *blast*
**moreover obtain** *y* **where** *y* ∈ *set r0 y* →$^*{}_T$ *x* **using** *assms*(*2*) *x-def* **by** *auto*
**ultimately show** *?thesis* **using** *reachable-reachable1-trans* **by** *blast*
**qed**(*simp*)

**lemma** *dverts-reachable1-if-dom-children-aux*:
⟦∀ *v*∈*dverts t1*. ∃ *x*∈*set r0* ∪ *X* ∪ *path-lverts t1* (*hd v*). *x* →$_T$ *hd v*;
∀ *y*∈*X*. ∃ *x*∈*set r0*. *x* →$^+{}_T$ *y*; ∀ *v*∈*dverts t1*. *forward v*; *v*∈*dverts t1*⟧
⟹ ∀ *y*∈*set v*. ∃ *x*∈*set r0*. *x* →$^+{}_T$ *y*
**proof**(*induction t1 arbitrary*: *X rule*: *dtree-to-list.induct*)
**case** (*1 r t e*)
**have** *r-reachable1*: ∀ *y*∈*set r*. ∃ *x*∈*set r0*. *x* →$^+{}_T$ *y*
**using** *dverts-reachable1-if-dom-children-aux-root*[*OF 1.prems*(*1*,*2*)] *1.prems*(*3*)
**by** *simp*
**then show** *?case*
**proof**(*cases r* = *v*)
**case** *True*
**then show** *?thesis* **using** *r-reachable1* **by** *simp*
**next**
**case** *False*
**have** *r-reach1*: ∀ *y*∈*set r* ∪ *X*. ∃ *x*∈*set r0*. *x* →$^+{}_T$ *y* **using** *1.prems*(*2*) *r-reachable1* **by** *blast*
**have** ∀ *x*. *path-lverts* (*Node r* {|(*t*, *e*)|}) *x* ⊆ *set r* ∪ *path-lverts t x*

```
    by simp
  then have 0: ∀ v∈dverts t. ∃ x∈set r0 ∪ (set r ∪ X) ∪ (path-lverts t (hd v)).
x →_T hd v
    using 1.prems(1) by fastforce
  then show ?thesis using 1.IH[OF 0 r-reach1] 1.prems(3,4) False by simp
 qed
next
 case (2 xs r)
 then show ?case
 proof(cases ∃ x∈set r0 ∪ X. x →_T hd v)
  case True
  then obtain x where x-def: x∈set r0 ∪ X x →_T hd v using 2.prems(1,4)
by blast
  then have hd v ∈ verts T using x-def(2) adj-in-verts(2) by auto
  moreover have forward v using 2.prems(3,4) by blast
  ultimately have v-reach1: ∀ y∈set v. x →^+_T y
   using hd-reach-all-forward x-def(2) reachable1-reachable-trans by blast
  then show ?thesis using 2.prems(2) x-def(1) reachable-reachable1-trans by
blast
 next
  case False
  then obtain x where x-def: x ∈ path-lverts (Node r xs) (hd v) x →_T hd v
   using 2.prems(1,4) by blast
  then have x ∈ set r using path-lverts.simps(2)[OF 2.hyps] empty-iff by metis
  then obtain x' where x'-def: x'∈set r0 x' →^+_T x
  using dverts-reachable1-if-dom-children-aux-root[OF 2.prems(1,2)] 2.prems(3)
by auto
  then have x'-v: x' →^+_T hd v using x-def(2) by simp
  then have hd v ∈ verts T using x-def(2) adj-in-verts(2) by auto
  moreover have forward v using 2.prems(3,4) by blast
  ultimately have v-reach1: ∀ y∈set v. x' →^+_T y
   using hd-reach-all-forward x'-v reachable1-reachable-trans by blast
  then show ?thesis using x'-def(1) by blast
 qed
qed

lemma dlverts-reachable1-if-dom-children-aux:
 ⟦∀ v∈dverts t1. ∃ x∈set r ∪ X ∪ path-lverts t1 (hd v). x →_T hd v;
   ∀ y∈X. ∃ x∈set r. x →^+_T y; ∀ v∈dverts t1. forward v; y∈dlverts t1⟧
  ⟹ ∃ x∈set r. x →^+_T y
 using dverts-reachable1-if-dom-children-aux list-in-verts-iff-lverts[of y t1] by blast

lemma dverts-reachable1-if-dom-children:
  assumes dom-children t1 T and v ∈ dverts t1 and v ≠ Dtree.root t1 and
∀ v∈dverts t1. forward v
 shows ∀ y∈set v. ∃ x∈set (Dtree.root t1). x →^+_T y
proof −
 obtain t2 where t2-def: t2 ∈ fst ‘ fset (sucs t1) v ∈ dverts t2
  using assms(2,3) dverts-root-or-suc by force
```

**then have** *0*: $\forall v{\in}$*dverts t2*. $\exists x{\in}$*set* (*Dtree.root t1*) $\cup$ {} $\cup$ *path-lverts t2* (*hd v*). $x \rightarrow_T$ *hd v*
**using** *assms*(*1*) **unfolding** *dom-children-def* **by** *blast*
**moreover have** $\forall v{\in}$*dverts t2*. *forward v* **using** *assms*(*4*) *t2-def*(*1*) *dverts-suc-subseteq* **by** *blast*
**ultimately show** *?thesis* **using** *dverts-reachable1-if-dom-children-aux t2-def*(*2*) **by** *blast*
**qed**

**lemma** *subtree-dverts-reachable1-if-mdeg-gt1*:
$[\![$*is-subtree t1 t*; *max-deg t1* $>$ *1*; $v \in$ *dverts t1*; $v \neq$ *Dtree.root t1*$]\!]$
$\Longrightarrow \forall y{\in}$*set v*. $\exists x{\in}$*set* (*Dtree.root t1*). $x \rightarrow^+{}_T\ y$
**proof**(*induction t1*)
**case** (*Node r xs*)
**then obtain** *t2 e2* **where** *t2-def*: (*t2*,*e2*) $\in$ *fset xs v* $\in$ *dverts t2* **by** *auto*
**then obtain** *x* **where** *x-def*: *x*$\in$*set r x* $\rightarrow_T$ *hd* (*Dtree.root t2*)
**using** *dom-mdeg-gt1 Node.prems*(*1*,*2*) **by** *fastforce*
**then have** *t2-T*: *hd* (*Dtree.root t2*) $\in$ *verts T* **using** *adj-in-verts*(*2*) **by** *simp*
**have** *is-subtree t2* (*Node r xs*) **using** *subtree-if-child*[*of t2 xs r*] *t2-def*(*1*) **by** *force*
**then have** *subt2*: *is-subtree t2 t* **using** *subtree-trans Node.prems*(*1*) **by** *blast*
**have** *Dtree.root t2* $\in$ *dverts t*
**using** *subt2 dverts-subtree-subset* **by** (*fastforce simp*: *dtree.set-sel*(*1*))
**then have** *fwd-t2*: *forward* (*Dtree.root t2*) **by** (*simp add*: *verts-forward*)
**then have** *t2-reach1*: $\forall y{\in}$*set* (*Dtree.root t2*). $x \rightarrow^+{}_T\ y$
**using** *hd-reach-all-forward*[*OF t2-T fwd-t2*] *x-def*(*2*) *reachable1-reachable-trans* **by** *blast*
**then consider** *Dtree.root t2* $=$ *v* | *Dtree.root t2* $\neq$ *v max-deg t2* $>$ *1* | *Dtree.root t2* $\neq$ *v max-deg t2* $\leq$ *1*
**by** *fastforce*
**then show** *?case*
**proof**(*cases*)
**case** *1*
**then show** *?thesis* **using** *t2-reach1 x-def*(*1*) **by** *auto*
**next**
**case** *2*
**then have** $\forall y{\in}$*set v*. $\exists x{\in}$*set* (*Dtree.root t2*). $x \rightarrow^+{}_T\ y$ **using** *Node.IH subt2 t2-def* **by** *simp*
**then show** *?thesis*
**using** *t2-reach1 x-def*(*1*) *reachable1-reachable reachable1-reachable-trans*
**unfolding** *dtree.sel*(*1*) **by** *blast*
**next**
**case** *3*
**then have** *fcard xs* $>$ *1* **using** *Node.prems*(*2*) *t2-def*(*1*) *fcard-gt1-if-mdeg-gt-child1* **by** *fastforce*
**then have** *dom*: *dom-children* (*Node r* {|(*t2*,*e2*)|}) *T*
**using** *dom-wedge-singleton*[*OF Node.prems*(*1*)] *t2-def*(*1*) *3*(*2*) **by** *fastforce*
**have** $\forall v \in$ *dverts* (*Node r xs*). *forward v*
**using** *Node.prems*(*1*) *seq-conform-alt verts-conform-subtree* **by** *blast*
**then have** $\forall v \in$ *dverts* (*Node r* {|(*t2*, *e2*)|}). *forward v* **using** *t2-def*(*1*) **by**

```
simp
    then show ?thesis
      using dverts-reachable1-if-dom-children[OF dom] t2-def(2) Node.prems(4)
      unfolding dtree.sel(1) by simp
  qed
qed

lemma subtree-dverts-reachable1-if-mdeg-gt1-singleton:
  assumes is-subtree (Node r {|(t1,e1)|}) t
      and max-deg (Node r {|(t1,e1)|}) > 1
      and v ∈ dverts t1
      and v ≠ Dtree.root t1
    shows ∀ y∈set v. ∃ x∈set (Dtree.root t1). x →⁺_T y
proof −
  have is-subtree t1 t using subtree-trans[OF subtree-if-child assms(1)] by simp
  then show ?thesis
    using assms(2−4) mdeg-eq-child-if-singleton-gt1[OF assms(2)]
      subtree-dverts-reachable1-if-mdeg-gt1 by simp
qed

lemma subtree-dverts-reachable1-if-mdeg-le1-subcontr:
  ⟦is-subtree t1 t; max-deg t1 ≤ 1; is-subtree (Node v2 {|(t2,e2)|}) t1;
    rank (rev (Dtree.root t2)) < rank (rev v2); v ∈ dverts t1; v ≠ Dtree.root t1⟧
    ⟹ ∀ y∈set v. ∃ x∈set (Dtree.root t1). x →⁺_T y
proof(induction t1)
  case (Node r xs)
  then show ?case
  proof(cases Node v2 {|(t2,e2)|} = Node r xs)
    case True
    then have dom-children (Node r xs) T using dom-contr′ Node.prems(1,2,4)
by blast
    moreover have ∀ v ∈ dverts (Node r xs). forward v
      using Node.prems(1) seq-conform-alt verts-conform-subtree by blast
   ultimately show ?thesis using dverts-reachable1-if-dom-children Node.prems(5,6)
by blast
  next
    case False
   then obtain t3 e3 where t3-def: (t3,e3) ∈ fset xs is-subtree (Node v2 {|(t2,e2)|})
t3
      using Node.prems(3) by auto
    then have t3-xs: xs = {|(t3,e3)|}
      using Node.prems(2) by (simp add: singleton-if-mdeg-le1-elem)
    then have v-t3: v ∈ dverts t3 using Node.prems(5,6) by simp
    then have t3-dom: ∃ x∈set r. x →_T hd (Dtree.root t3)
      using dom-sub-contr Node.prems(1,3,4) t3-xs by fastforce
    then have t3-T: hd (Dtree.root t3) ∈ verts T using adj-in-verts(2) by blast
    have is-subtree t3 (Node r xs) using subtree-if-child[of t3 xs] t3-xs by simp
    then have sub-t3: is-subtree t3 t using subtree-trans Node.prems(1) by blast
    then have Dtree.root t3 ∈ dverts t
```

```
    using dverts-subtree-subset by (fastforce simp: dtree.set-sel(1))
   then have forward (Dtree.root t3) by (simp add: verts-forward)
   then have t3-reach1: ∃ x∈set r. ∀ y∈set(Dtree.root t3). x →⁺_T y
     using hd-reach-all-forward[OF t3-T] t3-dom reachable1-reachable-trans by
blast
   show ?thesis
   proof(cases v = Dtree.root t3)
    case True
    then show ?thesis using t3-reach1 by auto
   next
    case False
   moreover have max-deg t3 ≤ 1 using Node.prems(2) t3-def(1) mdeg-ge-child
by fastforce
    ultimately have ∀ y∈set v. ∃ x∈set (Dtree.root t3). x →⁺_T y
     using Node.IH sub-t3 t3-def Node.prems(4) v-t3 by simp
    then show ?thesis
     using t3-reach1 reachable1-reachable-trans reachable1-reachable unfolding
dtree.sel(1)
     by blast
   qed
  qed
qed

lemma subtree-y-reach-if-mdeg-gt1-notroot-reach:
  assumes is-subtree (Node r {|(t1,e1)|}) t
    and max-deg (Node r {|(t1,e1)|}) > 1
    and v ≠ r
    and v ∈ dverts t
    and v ≠ Dtree.root t1
    and y ∈ set v
    and ∃ x∈set r. x →⁺_T y
   shows ∃ x'∈set (Dtree.root t1). x' →⁺_T y
proof −
 have v ∈ dverts (Node r {|(t1,e1)|}) using dverts-reach1-in-dverts-r assms(1,4,6,7)
by blast
 then show ?thesis using subtree-dverts-reachable1-if-mdeg-gt1-singleton assms(1−3,5,6)
by simp
qed

lemma subtree-eqroot-if-mdeg-gt1-reach:
  ⟦is-subtree (Node r {|(t1,e1)|}) t; max-deg (Node r {|(t1,e1)|}) > 1; v ∈ dverts
t;
   ∃ y∈set v. ¬(∃ x'∈set (Dtree.root t1). x' →⁺_T y) ∧ (∃ x∈set r. x →⁺_T y); v ≠
r⟧
   ⟹ Dtree.root t1 = v
  using subtree-y-reach-if-mdeg-gt1-notroot-reach by blast

lemma subtree-rank-ge-if-mdeg-gt1-reach:
  ⟦is-subtree (Node r {|(t1,e1)|}) t; max-deg (Node r {|(t1,e1)|}) > 1; v ∈ dverts
```

```
t;
    ∃ y∈set v. ¬(∃ x'∈set (Dtree.root t1). x' →⁺_T y) ∧ (∃ x∈set r. x →⁺_T y); v ≠
r⟧
    ⟹ rank (rev (Dtree.root t1)) ≤ rank (rev v)
  using subtree-eqroot-if-mdeg-gt1-reach by blast

lemma subtree-y-reach-if-mdeg-le1-notroot-subcontr:
  assumes is-subtree (Node r {|(t1,e1)|}) t
      and max-deg (Node r {|(t1,e1)|}) ≤ 1
      and is-subtree (Node v2 {|(t2,e2)|}) t1
      and rank (rev (Dtree.root t2)) < rank (rev v2)
      and v ≠ r
      and v ∈ dverts t
      and v ≠ Dtree.root t1
      and y ∈ set v
      and ∃ x∈set r. x →⁺_T y
    shows ∃ x'∈set (Dtree.root t1). x' →⁺_T y
proof −
  have 0: is-subtree t1 (Node r {|(t1,e1)|}) using subtree-if-child[of t1 {|(t1,e1)|}]
by simp
  then have subt1: is-subtree t1 t using assms(1) subtree-trans by blast
  have v ∈ dverts (Node r {|(t1,e1)|})
    using dverts-reach1-in-dverts-r assms(1,6,8,9) by blast
  then have v ∈ dverts t1 using assms(5) by simp
  moreover have max-deg t1 ≤ 1 using assms(2) mdeg-ge-sub[OF 0] by simp
  ultimately show ?thesis
    using subtree-dverts-reachable1-if-mdeg-le1-subcontr[OF subt1] assms(3,4,7,8)
by blast
qed

lemma rank-ge-if-mdeg-le1-dvert-nocontr:
  assumes max-deg t1 ≤ 1
      and ∄ v2 t2 e2. is-subtree (Node v2 {|(t2,e2)|}) t1 ∧ rank (rev (Dtree.root
t2)) < rank (rev v2)
      and v ∈ dverts t1
    shows rank (rev (Dtree.root t1)) ≤ rank (rev v)
using assms proof(induction t1)
  case (Node r xs)
  then show ?case
  proof(cases v = r)
    case False
    then obtain t2 e2 where t2-def: xs = {|(t2,e2)|} v ∈ dverts t2
      using Node.prems(1,3) singleton-if-mdeg-le1-elem by fastforce
   have max-deg t2 ≤ 1 using Node.prems(1) mdeg-ge-child[of t2 e2 xs] t2-def(1)
by simp
    then have rank (rev (Dtree.root t2)) ≤ rank (rev v)
      using Node.IH t2-def Node.prems(2) by fastforce
    then show ?thesis using Node.prems(2) t2-def(1) by fastforce
  qed(simp)
```

**qed**

**lemma** *subtree-rank-ge-if-mdeg-le1-nocontr*:
**assumes** *is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t*
**and** *max-deg* (*Node r* {|(*t1*,*e1*)|}) ≤ *1*
**and** ∄ *v2 t2 e2*. *is-subtree* (*Node v2* {|(*t2*,*e2*)|}) *t1* ∧ *rank* (*rev* (*Dtree.root t2*)) < *rank* (*rev v2*)
**and** *v* ≠ *r*
**and** *v* ∈ *dverts t*
**and** *y* ∈ *set v*
**and** ∃ *x*∈*set r*. *x* →⁺ *T y*
**shows** *rank* (*rev* (*Dtree.root t1*)) ≤ *rank* (*rev v*)
**proof** −
**have** *0*: *is-subtree t1* (*Node r* {|(*t1*,*e1*)|}) **using** *subtree-if-child*[*of t1* {|(*t1*,*e1*)|}] **by** *simp*
**then have** *0*: *max-deg t1* ≤ *1* **using** *assms*(*2*) *mdeg-ge-sub*[*OF 0*] **by** *simp*
**have** *v* ∈ *dverts* (*Node r* {|(*t1*,*e1*)|}) **using** *dverts-reach1-in-dverts-r assms*(*1*,*5*−*7*) **by** *blast*
**then have** *v* ∈ *dverts t1* **using** *assms*(*4*) **by** *simp*
**then show** *?thesis* **using** *rank-ge-if-mdeg-le1-dvert-nocontr 0 assms*(*3*) **by** *blast*
**qed**

**lemma** *subtree-rank-ge-if-mdeg-le1′*:
⟦*is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t*; *max-deg* (*Node r* {|(*t1*,*e1*)|}) ≤ *1*; *v* ≠ *r*;
*v* ∈ *dverts t*; *y* ∈ *set v*; ∃ *x*∈*set r*. *x* →⁺ *T y*; ¬(∃ *x′*∈*set* (*Dtree.root t1*). *x′* →⁺ *T y*)⟧
⟹ *rank* (*rev* (*Dtree.root t1*)) ≤ *rank* (*rev v*)
**using** *subtree-y-reach-if-mdeg-le1-notroot-subcontr subtree-rank-ge-if-mdeg-le1-nocontr*
**apply**(*cases* ∃ *v2 t2 e2*. *is-subtree* (*Node v2* {|(*t2*,*e2*)|}) *t1* ∧ *rank* (*rev* (*Dtree.root t2*))<*rank* (*rev v2*))
**by** *blast*+

**lemma** *subtree-rank-ge-if-mdeg-le1*:
⟦*is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t*; *max-deg* (*Node r* {|(*t1*,*e1*)|}) ≤ *1*; *v* ≠ *r*;
*v* ∈ *dverts t*; ∃ *y* ∈ *set v*. ¬(∃ *x′*∈*set* (*Dtree.root t1*). *x′* →⁺ *T y*) ∧ (∃ *x*∈*set r*. *x* →⁺ *T y*)⟧
⟹ *rank* (*rev* (*Dtree.root t1*)) ≤ *rank* (*rev v*)
**using** *subtree-y-reach-if-mdeg-le1-notroot-subcontr subtree-rank-ge-if-mdeg-le1-nocontr*
**apply**(*cases* ∃ *v2 t2 e2*. *is-subtree* (*Node v2* {|(*t2*,*e2*)|}) *t1* ∧ *rank* (*rev* (*Dtree.root t2*))<*rank* (*rev v2*))
**by** *blast*+

**lemma** *subtree-rank-ge-if-reach*:
⟦*is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t*; *v* ≠ *r*; *v* ∈ *dverts t*;
∃ *y* ∈ *set v*. ¬(∃ *x′*∈*set* (*Dtree.root t1*). *x′* →⁺ *T y*) ∧ (∃ *x*∈*set r*. *x* →⁺ *T y*)⟧
⟹ *rank* (*rev* (*Dtree.root t1*)) ≤ *rank* (*rev v*)
**using** *subtree-rank-ge-if-mdeg-le1 subtree-rank-ge-if-mdeg-gt1-reach*
**by** (*cases max-deg* (*Node r* {|(*t1*,*e1*)|}) ≤ *1*) (*auto simp del*: *max-deg.simps*)

**lemma** *subtree-rank-ge-if-reach'*:
*is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t* ⟹ ∀ *v* ∈ *dverts t*.
(∃ *y*∈*set v*. ¬ (∃ *x'*∈*set* (*Dtree.root t1*). *x'* →⁺$_T$ *y*) ∧ (∃ *x*∈*set r*. *x* →⁺$_T$ *y*) ∧ *v* ≠ *r*)
⟶ *rank* (*rev* (*Dtree.root t1*)) ≤ *rank* (*rev v*)
**using** *subtree-rank-ge-if-reach* **by** *blast*

### 10.3.1 Normalizing preserves Arc Invariants

**lemma** *normalize1-mdeg-le*: *max-deg* (*normalize1 t1*) ≤ *max-deg t1*
**proof**(*induction t1 rule*: *normalize1.induct*)
**case** (*1 r t e*)
**then show** *?case*
**proof**(*cases rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*))
**case** *True*
**then show** *?thesis* **using** *mdeg-child-sucs-le* **by** *fastforce*
**next**
**case** *False*
**then have** *max-deg* (*normalize1* (*Node r* {|(*t*, *e*)|}))
= *max* (*max-deg* (*normalize1 t*)) (*fcard* {|(*normalize1 t*, *e*)|})
**using** *mdeg-singleton* **by** *force*
**then show** *?thesis* **using** *mdeg-singleton*[*of r t*] *1 False* **by** (*simp add*: *fcard-single-1*)
**qed**
**next**
**case** (*2 xs r*)
**then have** *0*: ∀ (*t*,*e*) ∈ *fset xs*. *max-deg* (*normalize1 t*) ≤ *max-deg t* **by** *fastforce*
**have** *max-deg* (*normalize1* (*Node r xs*)) = *max-deg* (*Node r* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs*))
**using** *2.hyps* **by** *simp*
**then show** *?case* **using** *mdeg-img-le'*[*OF 0*] **by** *simp*
**qed**

**lemma** *normalize1-mdeg-eq*:
*wf-darcs t1*
⟹ *max-deg* (*normalize1 t1*) = *max-deg t1* ∨ (*max-deg* (*normalize1 t1*) = *0* ∧ *max-deg t1* = *1*)
**proof**(*induction t1 rule*: *normalize1.induct*)
**case** *ind*: (*1 r t e*)
**then have** *0*: *max-deg* (*Node r* {|(*t*, *e*)|}) ≥ *1*
**using** *mdeg-ge-fcard*[*of* {|(*t*, *e*)|}] **by** (*simp add*: *fcard-single-1*)
**then consider** *rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*)
| ¬*rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*) *max-deg* (*normalize1 t*) ≤ *1*
| ¬*rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*) *max-deg* (*normalize1 t*) > *1* **by** *linarith*
**then show** *?case*
**proof**(*cases*)
**case** *1*
**then show** *?thesis*
**using** *mdeg-singleton mdeg-root fcard-single-1*

```
    by (metis max-def nle-le dtree.exhaust-sel leI less-one normalize1.simps(1))
  next
    case 2
    then have max-deg (normalize1 (Node r {|(t, e)|})) = 1
      using mdeg-singleton[of r normalize1 t] by (auto simp: fcard-single-1)
    moreover have max-deg (Node r {|(t, e)|}) = 1
      using mdeg-singleton[of r t] ind 2
      by (auto simp: fcard-single-1 wf-darcs-iff-darcs')
    ultimately show ?thesis by simp
  next
    case 3
    then show ?thesis
      using mdeg-singleton[of r t] mdeg-singleton[of r normalize1 t] ind
      by (auto simp: fcard-single-1)
  qed
next
  case ind: (2 xs r)
  then consider max-deg (Node r xs) ≤ 1
    | max-deg (Node r xs) > 1 max-deg (Node r xs) = fcard xs
    | max-deg (Node r xs) > 1 fcard xs < max-deg (Node r xs)
    using mdeg-ge-fcard[of xs] by fastforce
  then show ?case
  proof(cases)
    case 1
    then show ?thesis using normalize1-mdeg-le[of Node r xs] by fastforce
  next
    case 2
    then have max-deg (Node r xs) ≤ max-deg (normalize1 (Node r xs))
      using mdeg-ge-fcard[of (λ(t, e). (normalize1 t, e)) |‘| xs] ind
      by (simp add: fcard-normalize-img-if-disjoint wf-darcs-iff-darcs')
    then show ?thesis using normalize1-mdeg-le[of Node r xs] by simp
  next
    case 3
    then obtain t e where t-def: (t,e) ∈ fset xs max-deg (Node r xs) = max-deg t
      using mdeg-child-if-gt-fcard by fastforce
    have max-deg (normalize1 t) ≤ max-deg (Node r ((λ(t,e). (normalize1 t,e)) |‘|
xs))
      using mdeg-ge-child[of normalize1 t e (λ(t,e). (normalize1 t,e)) |‘| xs r]
t-def(1)
      by fastforce
    then have max-deg (Node r xs) ≤ max-deg (normalize1 (Node r xs))
      using ind.hyps ind.IH[OF t-def(1) refl] ind.prems 3(1) t-def
      by (fastforce simp: wf-darcs-iff-darcs')
    then show ?thesis using normalize1-mdeg-le[of Node r xs] by simp
  qed
qed

lemma normalize1-mdeg-eq':
  wf-dlverts t1
```

$\Longrightarrow$ *max-deg* (*normalize1 t1*) = *max-deg t1* $\vee$ (*max-deg* (*normalize1 t1*) = *0* $\wedge$ *max-deg t1* = *1*)
**proof**(*induction t1 rule*: *normalize1.induct*)
**case** *ind*: (*1 r t e*)
**then have** *0*: *max-deg* (*Node r* {|(*t*, *e*)|}) $\geq$ *1*
**using** *mdeg-ge-fcard*[*of* {|(*t*, *e*)|}] **by** (*simp add*: *fcard-single-1*)
**then consider** *rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*)
| ¬*rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*) *max-deg* (*normalize1 t*) $\leq$ *1*
| ¬*rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*) *max-deg* (*normalize1 t*) > *1* **by** *linarith*
**then show** *?case*
**proof**(*cases*)
**case** *1*
**then show** *?thesis*
**using** *mdeg-singleton*[*of r t*] *mdeg-root*[*of Dtree.root t sucs t*]
**by** (*auto simp*: *fcard-single-1 simp del*: *max-deg.simps*)
**next**
**case** *2*
**then have** *max-deg* (*normalize1* (*Node r* {|(*t*, *e*)|})) = *1*
**using** *mdeg-singleton*[*of r normalize1 t*] **by** (*auto simp*: *fcard-single-1*)
**moreover have** *max-deg* (*Node r* {|(*t*, *e*)|}) = *1*
**using** *mdeg-singleton*[*of r t*] *ind 2* **by** (*auto simp*: *fcard-single-1*)
**ultimately show** *?thesis* **by** *simp*
**next**
**case** *3*
**then show** *?thesis*
**using** *mdeg-singleton*[*of r t*] *mdeg-singleton*[*of r normalize1 t*] *ind*
**by** (*auto simp*: *fcard-single-1*)
**qed**
**next**
**case** *ind*: (*2 xs r*)
**consider** *max-deg* (*Node r xs*) $\leq$ *1*
| *max-deg* (*Node r xs*) > *1 max-deg* (*Node r xs*) = *fcard xs*
| *max-deg* (*Node r xs*) > *1 fcard xs* < *max-deg* (*Node r xs*)
**using** *mdeg-ge-fcard*[*of xs*] **by** *fastforce*
**then show** *?case*
**proof**(*cases*)
**case** *1*
**then show** *?thesis* **using** *normalize1-mdeg-le*[*of Node r xs*] **by** (*auto simp del*: *max-deg.simps*)
**next**
**case** *2*
**have** *0*: $\forall$(*t*, *e*)$\in$*fset xs*. *dlverts t* $\neq$ {} **using** *dlverts-nempty-if-wf ind.prems* **by** *auto*
**then have** *max-deg* (*Node r xs*) $\leq$ *max-deg* (*normalize1* (*Node r xs*))
**using** *mdeg-ge-fcard*[*of* ($\lambda$(*t*, *e*). (*normalize1 t*, *e*)) |`| *xs*] *ind 2*
**by** (*simp add*: *fcard-normalize-img-if-disjoint-lverts*)
**then show** *?thesis* **using** *normalize1-mdeg-le*[*of Node r xs*] **by** *simp*
**next**

```
    case 3
    then obtain t e where t-def: (t,e) ∈ fset xs max-deg (Node r xs) = max-deg t
      using mdeg-child-if-gt-fcard by fastforce
    have max-deg (normalize1 t) ≤ max-deg (Node r ((λ(t,e). (normalize1 t,e)) |`| xs))
     using mdeg-ge-child[of normalize1 t e (λ(t,e). (normalize1 t,e)) |`| xs] t-def(1)
      by (force simp del: max-deg.simps)
    then have max-deg (Node r xs) ≤ max-deg (normalize1 (Node r xs))
      using ind 3(1) t-def by (fastforce simp del: max-deg.simps)
    then show ?thesis using normalize1-mdeg-le[of Node r xs] by simp
  qed
qed

lemma normalize1-dom-mdeg-gt1:
  ⟦is-subtree (Node r xs) (normalize1 t); t1 ∈ fst ` fset xs; max-deg (Node r xs) > 1⟧
    ⟹ ∃ v ∈ set r. v →_T hd (Dtree.root t1)
using ranked-dtree-with-orig-axioms proof(induction t rule: normalize1.induct)
  case (1 r1 t e)
  then interpret R: ranked-dtree-with-orig Node r1 {|(t,e)|} by blast
  have sub-t: is-subtree t (Node r1 {|(t,e)|}) using subtree-if-child[of t {|(t,e)|}]
by simp
  show ?case
  proof(cases Node r xs = normalize1 (Node r1 {|(t,e)|}))
    case eq: True
    then have 0: max-deg (Node r1 {|(t,e)|}) > 1
      by (metis normalize1-mdeg-le 1.prems(3) less-le-trans)
   then have max-t: max-deg t > 1 by (metis dtree.exhaust-sel mdeg-child-sucs-eq-if-gt1)
    then show ?thesis
    proof(cases rank (rev (Dtree.root t)) < rank (rev r1))
      case True
     then have eq: Node r xs = Node (r1@Dtree.root t) (sucs t) using eq by simp
      then have t1 ∈ fst ` fset (sucs t) using 1.prems(2) by simp
      then obtain v where v ∈ set (Dtree.root t) v →_T hd (Dtree.root t1)
        using R.dom-mdeg-gt1[of Dtree.root t sucs t] sub-t max-t by auto
      then show ?thesis using eq by auto
    next
      case False
      obtain v where v-def: v ∈ set r1 v →_T hd (Dtree.root t)
        using max-t R.dom-mdeg-gt1[of r1 {|(t, e)|}] 0 by auto
      interpret T: ranked-dtree-with-orig t using R.ranked-dtree-orig-rec by simp
      have eq: Node r xs = Node r1 {|(normalize1 t, e)|} using False eq by simp
      then have t1 = normalize1 t using 1.prems(2) by simp
      moreover have Dtree.root t ≠ []
        using empty-notin-wf-dlverts[OF T.wf-lverts] dtree.set-sel(1)[of t] by auto
       ultimately have hd (Dtree.root t1) = hd (Dtree.root t) using normal-
ize1-hd-root-eq by blast
      then show ?thesis using v-def eq by auto
    qed
```

```
  next
    case uneq: False
    show ?thesis
    proof(cases rank (rev (Dtree.root t)) < rank (rev r1))
      case True
     then have normalize1 (Node r1 {|(t,e)|}) = Node (r1@Dtree.root t) (sucs t)
by simp
      then obtain t2 where t2-def: t2 ∈ fst ‘ fset (sucs t) is-subtree (Node r xs)
t2
        using uneq 1.prems(1) by fastforce
      then have is-subtree t2 t using subtree-if-suc by blast
      then have is-subtree (Node r xs) (Node r1 {|(t,e)|})
        using subtree-trans subtree-if-suc t2-def(2) by auto
      then show ?thesis using R.dom-mdeg-gt1 1.prems by blast
    next
      case False
     then have normalize1 (Node r1 {|(t,e)|}) = Node r1 {|(normalize1 t, e)|}
by simp
      then have is-subtree (Node r xs) (normalize1 t) using uneq 1.prems(1) by
auto
     then show ?thesis using 1.IH False 1.prems(2,3) R.ranked-dtree-orig-rec by
simp
    qed
  qed
next
  case (2 xs1 r1)
  then interpret R: ranked-dtree-with-orig Node r1 xs1 by blast
  show ?case
  proof(cases Node r xs = normalize1 (Node r1 xs1))
    case True
    then have 0: max-deg (Node r1 xs1) > 1
       using normalize1-mdeg-le 2.prems(3) less-le-trans by (fastforce simp del:
max-deg.simps)
    then obtain t where t-def: t ∈ fst ‘ fset xs1 normalize1 t = t1
      using 2.prems(2) 2.hyps True by fastforce
    then have sub-t: is-subtree t (Node r1 xs1) using subtree-if-child by fast
    then obtain v where v-def: v ∈ set r1 v →T hd (Dtree.root t)
      using R.dom-mdeg-gt1[of r1] t-def(1) 0 by auto
     interpret T: ranked-dtree-with-orig t using R.ranked-dtree-orig-rec t-def(1)
by force
    have Dtree.root t ≠ []
      using empty-notin-wf-dlverts[OF T.wf-lverts] dtree.set-sel(1)[of t] by auto
    then have hd (Dtree.root t1) = hd (Dtree.root t) using normalize1-hd-root-eq
t-def(2) by blast
    then show ?thesis using v-def 2.hyps True by auto
  next
    case False
    then show ?thesis using 2 R.ranked-dtree-orig-rec by auto
  qed
```

**qed**

**lemma** *child-contr-if-new-contr*:
**assumes** ¬*rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*)
**and** *rank* (*rev* (*Dtree.root* (*normalize1 t1*))) < *rank* (*rev r*)
**shows** ∃ *t2 e2*. *sucs t1* = {|(*t2*,*e2*)|} ∧ *rank* (*rev* (*Dtree.root t2*)) < *rank* (*rev* (*Dtree.root t1*))
**proof** −
**obtain** *t2 e2* **where** *t2-def*: *sucs t1* = {|(*t2*,*e2*)|}
**using** *root-normalize1-eq2*[*of sucs t1 Dtree.root t1*] *assms* **by** *fastforce*
**then show** *?thesis*
**using** *root-normalize1-eq1*[*of t2 Dtree.root t1 e2*] *assms dtree.collapse*[*of t1*] **by** *fastforce*
**qed**

**lemma** *sub-contr-if-new-contr*:
**assumes** ¬*rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*)
**and** *rank* (*rev* (*Dtree.root* (*normalize1 t1*))) < *rank* (*rev r*)
**shows** ∃ *v t2 e2*. *is-subtree* (*Node v* {|(*t2*,*e2*)|}) *t1* ∧ *rank* (*rev* (*Dtree.root t2*)) < *rank* (*rev v*)
**proof** −
**obtain** *t2 e2* **where** *t2-def*: *sucs t1* = {|(*t2*,*e2*)|} *rank* (*rev* (*Dtree.root t2*)) < *rank* (*rev* (*Dtree.root t1*))
**using** *child-contr-if-new-contr*[*OF assms*] **by** *blast*
**then have** *is-subtree* (*Node* (*Dtree.root t1*) {|(*t2*,*e2*)|}) *t1*
**using** *is-subtree.simps*[*of Node* (*Dtree.root t1*) {|(*t2*,*e2*)|} *Dtree.root t1 sucs t1*] **by** *fastforce*
**then show** *?thesis* **using** *t2-def*(*2*) **by** *blast*
**qed**

**lemma** *normalize1-subtree-same-hd*:
⟦*is-subtree* (*Node v* {|(*t1*,*e1*)|}) (*normalize1 t*)⟧
⟹ ∃ *t3 e3*. (*is-subtree* (*Node v* {|(*t3*,*e3*)|}) *t* ∧ *hd* (*Dtree.root t1*) = *hd* (*Dtree.root t3*))
∨ (∃ *v2*. *v* = *v2* @ *Dtree.root t3* ∧ *sucs t3* = {|(*t1*,*e1*)|}
∧ *is-subtree* (*Node v2* {|(*t3*,*e3*)|}) *t* ∧ *rank* (*rev* (*Dtree.root t3*)) < *rank* (*rev v2*))
**using** *wf-lverts wf-arcs* **proof**(*induction t rule*: *normalize1.induct*)
**case** (*1 r t e*)
**show** *?case*
**proof**(*cases Node v* {|(*t1*,*e1*)|} = *normalize1* (*Node r* {|(*t*,*e*)|}))
**case** *eq*: *True*
**then show** *?thesis*
**proof**(*cases rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*))
**case** *True*
**then show** *?thesis* **using** *1 eq* **by** *auto*
**next**
**case** *False*
**then have** *eq*: *Node v* {|(*t1*,*e1*)|} = *Node r* {|(*normalize1 t*,*e*)|} **using** *eq* **by**

```
simp
      then show ?thesis using normalize1-hd-root-eq' 1.prems(2) by auto
    qed
  next
    case uneq: False
    then show ?thesis
    proof(cases rank (rev (Dtree.root t)) < rank (rev r))
      case True
    then obtain t2 e2 where (t2,e2) ∈ fset (sucs t) is-subtree (Node v {|(t1,e1)|}) t2
        using 1.prems(1) uneq by auto
      then show ?thesis using is-subtree.simps[of Node v {|(t1,e1)|} Dtree.root t sucs t] by auto
    next
      case False
      then have is-subtree (Node v {|(t1,e1)|}) (normalize1 t) using 1.prems(1) uneq by auto
      then show ?thesis
        using 1.IH 1.prems(2,3) False by (auto simp: wf-darcs-iff-darcs')
    qed
  qed
next
  case (2 xs r)
  then have ∀ x. ((λ(t,e). (normalize1 t,e)) |`| xs) ≠ {|x|}
    using singleton-normalize1 by (simp add: wf-darcs-iff-darcs')
  then have Node v {|(t1,e1)|} ≠ Node r ((λ(t,e). (normalize1 t,e)) |`| xs) by auto
  then obtain t2 e2 where (t2,e2) ∈ fset xs ∧ is-subtree (Node v {|(t1,e1)|}) (normalize1 t2)
    using 2.prems(1) 2.hyps by auto
 then show ?case using 2.IH 2.prems(2,3) by (fastforce simp: wf-darcs-iff-darcs')
qed

lemma normalize1-dom-sub-contr:
  ⟦is-subtree (Node r xs) (normalize1 t); t1 ∈ fst ` fset xs;
    ∃ v t2 e2. is-subtree (Node v {|(t2,e2)|}) (Node r xs) ∧ rank (rev (Dtree.root t2)) < rank (rev v)⟧
   ⟹ ∃ v ∈ set r. v →T hd (Dtree.root t1)
using ranked-dtree-with-orig-axioms proof(induction t rule: normalize1.induct)
  case (1 r1 t e)
  then interpret R: ranked-dtree-with-orig Node r1 {|(t,e)|} by blast
  interpret T: ranked-dtree-with-orig t using R.ranked-dtree-orig-rec by simp
  have sub-t: is-subtree (Node (Dtree.root t) (sucs t)) (Node r1 {|(t,e)|})
    using subtree-if-child[of t {|(t,e)|}] by simp
  obtain v t2 e2 where v-def:
      is-subtree (Node v {|(t2,e2)|}) (Node r xs) rank (rev (Dtree.root t2)) < rank (rev v)
    using 1.prems(3) by blast
  show ?case
```

**proof**(*cases Node r xs = normalize1 (Node r1 {|(t,e)|})*)
**case** *eq*: *True*
**then show** *?thesis*
**proof**(*cases rank (rev (Dtree.root t)) < rank (rev r1)*)
**case** *True*
**then have** *eq*: *Node r xs = Node (r1@Dtree.root t) (sucs t)* **using** *eq* **by** *simp*
**then consider** *Node r xs = Node v {|(t2,e2)|} max-deg (Node r xs) ≤ 1*
| *Node r xs ≠ Node v {|(t2,e2)|} | max-deg (Node r xs) > 1*
**by** *linarith*
**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then have** *max-deg (Node (r1@Dtree.root t) (sucs t)) ≤ 1* **using** *eq* **by** *blast*
**then have** *max-deg t ≤ 1* **using** *mdeg-root*[*of Dtree.root t sucs t*] **by** *simp*
**then have** *max-deg (Node r1 {|(t,e)|}) = 1*
**using** *mdeg-singleton*[*of r1 t*] **by** (*simp add*: *fcard-single-1*)
**then have** *dom*: *dom-children (Node r1 {|(t, e)|}) T* **using** *R.dom-contr True* **by** *auto*
**have** *0*: *t1 ∈ fst ‘ fset (sucs t)* **using** *eq 1.prems*(*2*) **by** *blast*
**then have** *Dtree.root t1 ∈ dverts t*
**using** *dtree.set-sel*(*1*) *T.dverts-child-subset dtree.exhaust-sel psubsetD* **by** *metis*
**then obtain** *r2* **where** *r2-def*:
*r2 ∈ set r1 ∪ path-lverts t (hd (Dtree.root t1)) r2 →T (hd (Dtree.root t1))*
**using** *dom* **unfolding** *dom-children-def* **by** *auto*
**have** *Dtree.root t1 ≠ []*
**using** *empty-notin-wf-dlverts T.wf-lverts 0 T.dverts-child-subset*
**by** (*metis dtree.exhaust-sel dtree.set-sel*(*1*) *psubsetD*)
**then have** *r2 ∈ set r1 ∪ set (Dtree.root t)*
**using** *path-lverts-subset-root-if-childhd*[*OF 0*] *r2-def*(*1*) **by** *fast*
**then show** *?thesis* **using** *r2-def*(*2*) *eq* **by** *auto*
**next**
**case** *2*
**then obtain** *t3 e3* **where** *t3-def*:
*(t3,e3) ∈ fset (sucs t) is-subtree (Node v {|(t2,e2)|}) t3*
**using** *eq v-def*(*1*) **by** *auto*
**have** *is-subtree t3 t* **using** *t3-def*(*1*) *subtree-if-suc* **by** *fastforce*
**then have** *is-subtree (Node v {|(t2,e2)|}) (Node (Dtree.root t) (sucs t))*
**using** *t3-def*(*2*) *subtree-trans* **by** *auto*
**moreover have** *t1 ∈ fst ‘ fset (sucs t)* **using** *eq 1.prems*(*2*) **by** *blast*
**ultimately obtain** *v* **where** *v-def*: *v ∈ set (Dtree.root t) ∧ v →T hd (Dtree.root t1)*
**using** *R.dom-sub-contr*[*OF sub-t*] *v-def*(*2*) *eq* **by** *blast*
**then show** *?thesis* **using** *eq* **by** *auto*
**next**
**case** *3*
**then show** *?thesis* **using** *R.normalize1-dom-mdeg-gt1 1.prems*(*1*,*2*) **by** *blast*

```
    qed
  next
    case False
    then have eq: Node r xs = Node r1 {|(normalize1 t, e)|} using eq by simp
    have hd: hd (Dtree.root (normalize1 t)) = hd (Dtree.root t)
      using normalize1-hd-root-eq' T.wf-lverts by blast
    have ∃ v t2 e2. is-subtree (Node v {|(t2,e2)|}) t ∧ rank (rev (Dtree.root t2))
< rank (rev v)
      using contr-before-normalize1 eq v-def sub-contr-if-new-contr False by auto
    then show ?thesis using R.dom-sub-contr[of r1 {|(t,e)|}] eq 1.prems(2) hd
by auto
  qed
 next
  case uneq: False
  show ?thesis
  proof(cases rank (rev (Dtree.root t)) < rank (rev r1))
    case True
   then have normalize1 (Node r1 {|(t,e)|}) = Node (r1@Dtree.root t) (sucs t)
by simp
    then obtain t2 where t2-def: t2 ∈ fst ‘ fset (sucs t) is-subtree (Node r xs)
t2
      using uneq 1.prems(1) by fastforce
    then have is-subtree t2 t using subtree-if-suc by blast
    then have is-subtree (Node r xs) (Node r1 {|(t,e)|})
      using subtree-trans subtree-if-child t2-def(2) by auto
    then show ?thesis using R.dom-sub-contr 1.prems(2,3) by fast
  next
    case False
     then have normalize1 (Node r1 {|(t,e)|}) = Node r1 {|(normalize1 t, e)|}
by simp
    then have is-subtree (Node r xs) (normalize1 t) using uneq 1.prems(1) by
auto
   then show ?thesis using 1.IH False 1.prems(2,3) R.ranked-dtree-orig-rec by
simp
  qed
 qed
next
 case (2 xs1 r1)
 then interpret R: ranked-dtree-with-orig Node r1 xs1 by blast
 show ?case
 proof(cases Node r xs = normalize1 (Node r1 xs1))
  case True
  then have eq: Node r xs = Node r1 ((λ(t,e). (normalize1 t,e)) |‘| xs1) using
2.hyps by simp
  obtain v t2 e2 where v-def:
    is-subtree (Node v {|(t2,e2)|}) (Node r xs) rank (rev (Dtree.root t2)) < rank
(rev v)
    using 2.prems(3) by blast
  obtain t where t-def: t ∈ fst ‘ fset xs1 normalize1 t = t1 using 2.prems(2)
```

*eq* **by** *force*
**then interpret** *T*: *ranked-dtree-with-orig t* **using** *R.ranked-dtree-orig-rec* **by** *force*
**have** ∃ *v t2 e2*. *is-subtree* (*Node v* {|(*t2*,*e2*)|}) (*Node r1 xs1*)
∧ *rank* (*rev* (*Dtree.root t2*)) < *rank* (*rev v*)
**using** *True contr-before-normalize1 v-def* **by** *presburger*
**moreover have** *hd* (*Dtree.root t1*) = *hd* (*Dtree.root t*)
**using** *normalize1-hd-root-eq′ T.wf-lverts t-def*(*2*) **by** *blast*
**ultimately show** *?thesis* **using** *R.dom-sub-contr*[*of r1 xs1*] *t-def*(*1*) *eq* **by** *auto*
**next**
**case** *False*
**then obtain** *t e* **where** (*t*,*e*) ∈ *fset xs1* ∧ *is-subtree* (*Node r xs*) (*normalize1 t*)
**using** *2.prems*(*1*) *2.hyps* **by** *auto*
**then show** *?thesis* **using** *2.IH 2.prems*(*2*,*3*) *R.ranked-dtree-orig-rec* **by** *fast*
**qed**
**qed**

**lemma** *dom-children-combine-aux*:
**assumes** *dom-children* (*Node r* {|(*t1*, *e1*)|}) *T*
**and** *t2* ∈ *fst* ‘ *fset* (*sucs t1*)
**and** *x* ∈ *dverts t2*
**shows** ∃ *v* ∈ *set* (*r* @ *Dtree.root t1*) ∪ *path-lverts t2* (*hd x*). *v* →$_T$ (*hd x*)
**using** *path-lverts-child-union-root-sub*[*OF assms*(*2*)] *assms dtree.set-sel*(*2*)
**unfolding** *dom-children-def* **by** *fastforce*

**lemma** *dom-children-combine*:
*dom-children* (*Node r* {|(*t1*, *e1*)|}) *T* ⟹ *dom-children* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*)) *T*
**using** *dom-children-combine-aux* **by** (*simp add*: *dom-children-def*)

**lemma** *path-lverts-normalize1-sub*:
⟦*wf-dlverts t1*; *x* ∈ *dverts* (*normalize1 t1*); *max-deg* (*normalize1 t1*) ≤ *1*⟧
⟹ *path-lverts t1* (*hd x*) ⊆ *path-lverts* (*normalize1 t1*) (*hd x*)
**proof**(*induction t1 rule*: *normalize1.induct*)
**case** (*1 r t e*)
**then show** *?case*
**proof**(*cases rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*))
**case** *True*
**then have** *eq*: *normalize1* (*Node r* {|(*t*, *e*)|}) = *Node* (*r*@*Dtree.root t*) (*sucs t*) **by** *simp*
**then show** *?thesis*
**proof**(*cases x* = *r*@*Dtree.root t*)
**case** *True*
**then show** *?thesis* **using** *1* **by** *auto*
**next**
**case** *False*
**then obtain** *t1 e1* **where** *t1-def*: (*t1*,*e1*) ∈ *fset* (*sucs t*) *x* ∈ *dverts t1*
**using** *1.prems*(*2*) *eq* **by** *auto*

```
    then have 0: hd x ∈ dlverts t1
      using hd-in-lverts-if-wf 1.prems(1) wf-dlverts-sucs by force
    then have hd x ∈ dlverts t using t1-def(1) suc-in-dlverts by fast
    then have 2: hd x ∉ set r using 1.prems(1) by auto
    have wf-dlverts t using 1.prems(1) by simp
     then have hd x ∉ set (Dtree.root t) using 0 t1-def(1) wf-dlverts.simps[of
Dtree.root t] by fastforce
    then have hd-nin: hd x ∉ set (r @ Dtree.root t) using 2 by auto
    then obtain t2 e2 where sucs t = {|(t2,e2)|}
     using 1.prems(3) ‹hd x ∈ dlverts t› ‹hd x ∉ set (Dtree.root t)› mdeg-root eq
       by (metis dtree.collapse denormalize.simps(2) denormalize-set-eq-dlverts
surj-pair)
    then show ?thesis using eq hd-nin path-lverts-simps1-sucs by fastforce
   qed
  next
   case uneq: False
    then have normalize1 (Node r {|(t, e)|}) = Node r {|(normalize1 t, e)|} by
simp
   then have max-deg (normalize1 t) ≤ 1
    using 1.prems(3) mdeg-singleton[of r normalize1 t] fcard-single-1 max-def by
auto
   then show ?thesis using uneq 1 by auto
  qed
next
  case (2 xs r)
  then have max-deg (normalize1 (Node r xs)) = max-deg (Node r xs) ∨ max-deg
(Node r xs) = 1
   using normalize1-mdeg-eq′ by blast
  then have max-deg (Node r xs) ≤ 1 using 2.prems(3) by (auto simp del:
max-deg.simps)
  then have fcard xs = 0
   using mdeg-ge-fcard[of xs r] fcard-single-1-iff[of xs] 2.hyps by fastforce
  then show ?case using 2 by simp
qed

lemma dom-children-normalize1-aux-1:
  assumes dom-children (Node r {|(t1, e1)|}) T
     and sucs t1 = {|(t2,e2)|}
     and wf-dlverts t1
     and normalize1 t1 = Node (Dtree.root t1 @ Dtree.root t2) (sucs t2)
     and max-deg t1 = 1
     and x ∈ dverts (normalize1 t1)
   shows ∃ v ∈ set r ∪ path-lverts (normalize1 t1) (hd x). v →T (hd x)
proof(cases x = Dtree.root t1 @ Dtree.root t2)
  case True
 then have 0: hd x = hd (Dtree.root t1) using assms(3,4) normalize1-hd-root-eq′
by fastforce
  then obtain v where v-def: v ∈ set r ∪ path-lverts t1 (hd x) v →T (hd x)
   using assms(1) dtree.set-sel(1) unfolding dom-children-def by auto
```

**have** *Dtree.root t1* $\neq$ [] **using** *assms*(*3*) *wf-dlverts.simps*[*of Dtree.root t1 sucs t1*]
**by** *simp*
**then show** *?thesis* **using** *v-def 0 path-lverts-empty-if-roothd* **by** *auto*
**next**
**case** *False*
**then obtain** *t3 e3* **where** *t3-def*: (*t3*,*e3*) $\in$ *fset* (*sucs t2*) *x* $\in$ *dverts t3*
**using** *assms*(*2*,*4*,*6*) **by** *auto*
**then have** *x* $\in$ *dverts t2* **using** *dtree.set*(*1*)[*of Dtree.root t2 sucs t2*] **by** *fastforce*
**then have** *x* $\in$ *dverts* (*Node* (*Dtree.root t1*) {|(*t2*,*e2*)|}) **by** *auto*
**then have** *x* $\in$ *dverts t1* **using** *assms*(*2*) *dtree.exhaust-sel* **by** *metis*
**then obtain** *v* **where** *v-def*: *v* $\in$ *set r* $\cup$ *path-lverts t1* (*hd x*) *v* $\rightarrow_T$ (*hd x*)
**using** *assms*(*1*) *dtree.set-sel*(*1*) **unfolding** *dom-children-def* **by** *auto*
**have** *path-lverts t1* (*hd x*) $\subseteq$ *path-lverts* (*Node* (*Dtree.root t1* @ *Dtree.root t2*)
(*sucs t2*)) (*hd x*)
**using** *assms*(*3*$-$*6*) *normalize1-mdeg-le path-lverts-normalize1-sub* **by** *metis*
**then show** *?thesis* **using** *v-def assms*(*4*) **by** *auto*
**qed**

**lemma** *dom-children-normalize1-1*:
⟦*dom-children* (*Node r* {|(*t1*, *e1*)|}) *T*; *sucs t1* = {|(*t2*,*e2*)|}; *wf-dlverts t1*;
*normalize1 t1* = *Node* (*Dtree.root t1* @ *Dtree.root t2*) (*sucs t2*); *max-deg t1* =
*1*⟧
$\Longrightarrow$ *dom-children* (*Node r* {|(*normalize1 t1*, *e1*)|}) *T*
**using** *dom-children-normalize1-aux-1* **by** (*simp add*: *dom-children-def*)

**lemma** *dom-children-normalize1-aux*:
**assumes** $\forall$ *x*$\in$*dverts t1*. $\exists$ *v* $\in$ *set r0* $\cup$ *path-lverts t1* (*hd x*). *v* $\rightarrow_T$ *hd x*
**and** *wf-dlverts t1*
**and** *max-deg t1* $\leq$ *1*
**and** *x* $\in$ *dverts* (*normalize1 t1*)
**shows** $\exists$ *v* $\in$ *set r0* $\cup$ *path-lverts* (*normalize1 t1*) (*hd x*). *v* $\rightarrow_T$ (*hd x*)
**using** *assms* **proof**(*induction t1 arbitrary*: *r0 rule*: *normalize1.induct*)
**case** (*1 r t e*)
**have** *deg1*: *max-deg* (*Node r* {|(*t*, *e*)|}) = *1*
**using** *1.prems*(*3*) *mdeg-ge-fcard*[*of* {|(*t*, *e*)|}] **by** (*simp add*: *fcard-single-1*)
**then show** *?case*
**proof**(*cases rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*))
**case** *True*
**have** *0*: *dom-children* (*Node r0* {|(*Node r* {|(*t*, *e*)|}, *e*)|}) *T*
**using** *1.prems*(*1*) **unfolding** *dom-children-def* **by** *simp*
**show** *?thesis* **using** *dom-children-normalize1-aux-1*[*OF 0*] *1.prems*(*1*,*2*,*4*) *deg1*
*True* **by** *auto*
**next**
**case** *ncontr*: *False*
**show** *?thesis*
**proof**(*cases x* = *r*)
**case** *True*
**then show** *?thesis* **using** *1.prems*(*1*,*2*) **by** *auto*
**next**

**case** *False*
**have** *wf-dlverts* (*normalize1 t*) **using** *1.prems*(*2*) *wf-dlverts-normalize1* **by** *auto*
**then have** *hd x* ∈ *dlverts* (*normalize1 t*)
**using** *hd-in-lverts-if-wf False ncontr 1.prems*(*1*,*4*) **by** *fastforce*
**then have** *hd*: *hd x* ∉ *set r* **using** *1.prems*(*2*) *ncontr wf-dlverts-normalize1* **by** *fastforce*
**then have** *eq*: *path-lverts* (*Node r* {|(*t*, *e*)|}) (*hd x*) = *set r* ∪ *path-lverts t* (*hd x*) **by** *simp*
**then have** *eq1*: *path-lverts* (*Node r* {|(*normalize1 t*, *e*)|}) (*hd x*)
= *set r* ∪ *path-lverts* (*normalize1 t*) (*hd x*) **by** *auto*
**have** ∀ *x*∈*dverts t*. *path-lverts* (*Node r* {|(*t*, *e*)|}) (*hd x*) ⊆ *set r* ∪ *path-lverts t* (*hd x*)
**using** *path-lverts-child-union-root-sub* **by** *simp*
**then have** *2*: ∀ *x*∈*dverts t*. ∃ *v*∈*set* (*r0*@*r*) ∪ *path-lverts t* (*hd x*). *v* →$_T$ *hd x*
**using** *1.prems*(*1*) **by** *fastforce*
**have** *max-deg t* ≤ *1* **using** *1.prems*(*3*) *mdeg-ge-child*[*of t e* {|(*t*, *e*)|}] **by** *simp*
**then show** *?thesis* **using** *1.IH*[*OF ncontr 2*] *1.prems*(*2*,*4*) *ncontr hd* **by** *auto*
**qed**
**qed**
**next**
**case** (*2 xs r*)
**then have** *fcard xs* ≤ *1* **using** *mdeg-ge-fcard*[*of xs*] **by** *simp*
**then have** *fcard xs* = *0* **using** *2.hyps fcard-single-1-iff*[*of xs*] **by** *fastforce*
**then show** *?case* **using** *2* **by** *auto*
**qed**

**lemma** *dom-children-normalize1*:
⟦*dom-children* (*Node r0* {|(*t1*,*e1*)|}) *T*; *wf-dlverts t1*; *max-deg t1* ≤ *1*⟧
⟹ *dom-children* (*Node r0* {|(*normalize1 t1*,*e1*)|}) *T*
**using** *dom-children-normalize1-aux* **by** (*simp add*: *dom-children-def*)

**lemma** *dom-children-child-self-aux*:
**assumes** *dom-children t1 T*
**and** *sucs t1* = {|(*t2*, *e2*)|}
**and** *rank* (*rev* (*Dtree.root t2*)) < *rank* (*rev* (*Dtree.root t1*))
**and** *t* = *Node r* {|(*t1*, *e1*)|}
**and** *x* ∈ *dverts t1*
**shows** ∃ *v* ∈ *set r* ∪ *path-lverts t1* (*hd x*). *v* →$_T$ *hd x*
**proof**(*cases x* = *Dtree.root t1*)
**case** *True*
**have** *is-subtree* (*Node* (*Dtree.root t1*) {|(*t2*, *e2*)|}) (*Node r* {|(*t1*, *e1*)|})
**using** *subtree-if-child*[*of t1* {|(*t1*, *e1*)|}] *assms*(*2*) *dtree.collapse*[*of t1*] **by** *simp*
**then show** *?thesis* **using** *dom-sub-contr*[*of r* {|(*t1*, *e1*)|}] *assms*(*3*,*4*) *True* **by** *auto*
**next**
**case** *False*
**then have** *x* ∈ (⋃ *y*∈*fset* (*sucs t1*). ⋃ (*dverts* ‘ *Basic-BNFs.fsts y*))
**using** *assms*(*5*) *dtree.set*(*1*)[*of Dtree.root t1 sucs t1*] **by** *auto*

**then have** $x \in$ *dverts t2* **using** *assms(2)* **by** *auto*
**then obtain** *v* **where** *v-def*: $v \in$ *set* (*Dtree.root t1*) $\cup$ *path-lverts t2* (*hd x*) *v* $\rightarrow_T$ (*hd x*)
**using** *assms(1,2) dtree.set-sel(1)* **unfolding** *dom-children-def* **by** *auto*
**interpret** *T1*: *list-dtree t1* **using** *list-dtree-rec assms(4)* **by** *simp*
**interpret** *T2*: *list-dtree t2* **using** *T1.list-dtree-rec-suc assms(2)* **by** *simp*
**have** *hd x* $\in$ *dlverts t2* **using** ‹$x \in$ *dverts t2*› **by** (*simp add*: *hd-in-lverts-if-wf T2.wf-lverts*)
**then have** *hd x* $\notin$ *set* (*Dtree.root t1*)
**using** *T1.wf-lverts wf-dlverts.simps*[*of Dtree.root t1 sucs t1*] *assms(2)* **by** *fastforce*
**then have** *path-lverts t1* (*hd x*) = *set* (*Dtree.root t1*) $\cup$ *path-lverts t2* (*hd x*)
**using** *assms(2)* **by** (*simp add*: *path-lverts-simps1-sucs*)
**then show** *?thesis* **using** *v-def* **by** *auto*
**qed**

**lemma** *dom-children-child-self*:
**assumes** *dom-children t1 T*
**and** *sucs t1* = {|(*t2*, *e2*)|}
**and** *rank* (*rev* (*Dtree.root t2*)) < *rank* (*rev* (*Dtree.root t1*))
**and** *t* = *Node r* {|(*t1*, *e1*)|}
**shows** *dom-children* (*Node r* {|(*t1*, *e1*)|}) *T*
**using** *dom-children-child-self-aux*[*OF assms*] **by** (*simp add*: *dom-children-def*)

**lemma** *normalize1-dom-contr*:
⟦*is-subtree* (*Node r* {|(*t1*,*e1*)|}) (*normalize1 t*); *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*);
*max-deg* (*Node r* {|(*t1*,*e1*)|}) = *1*⟧
⟹ *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T*
**using** *ranked-dtree-with-orig-axioms* **proof**(*induction t rule*: *normalize1.induct*)
**case** (*1 r1 t e*)
**then interpret** *R*: *ranked-dtree-with-orig Node r1* {|(*t*,*e*)|} **by** *blast*
**interpret** *T*: *ranked-dtree-with-orig t* **using** *R.ranked-dtree-orig-rec* **by** *simp*
**have** *sub-t*: *is-subtree* (*Node* (*Dtree.root t*) (*sucs t*)) (*Node r1* {|(*t*,*e*)|})
**using** *subtree-if-child*[*of t* {|(*t*,*e*)|}] **by** *simp*
**show** *?case*
**proof**(*cases Node r* {|(*t1*,*e1*)|} = *normalize1* (*Node r1* {|(*t*,*e*)|}))
**case** *eq*: *True*
**then show** *?thesis*
**proof**(*cases rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r1*))
**case** *True*
**then have** *eq*: *Node r* {|(*t1*,*e1*)|} = *Node* (*r1*@*Dtree.root t*) (*sucs t*) **using** *eq* **by** *simp*
**then have** *max-deg t* = *1* **using** *mdeg-root*[*of Dtree.root t sucs t*] *1* **by** *simp*
**then have** *max-deg* (*Node r1* {|(*t*,*e*)|}) = *1*
**using** *mdeg-singleton*[*of r1 t*] **by** (*simp add*: *fcard-single-1*)
**then have** *dom-children* (*Node r1* {|(*t*, *e*)|}) *T* **using** *R.dom-contr*[*of r1 t e*] *True* **by** *simp*
**then show** *?thesis* **using** *dom-children-combine eq* **by** *simp*

```
    next
      case False
      then have eq: Node r {|(t1,e1)|} = Node r1 {|(normalize1 t, e)|} using eq
by simp
      then obtain t2 e2 where t2-def:
          sucs t = {|(t2, e2)|} rank (rev (Dtree.root t2)) < rank (rev (Dtree.root t))
        using child-contr-if-new-contr False 1.prems(2) by blast
       then have is-subtree (Node (Dtree.root t) {|(t2, e2)|}) (Node r1 {|(t, e)|})
using sub-t by simp
      have max-deg t = 1
        using 1.prems(3) eq mdeg-singleton mdeg-root t2-def
        by (metis dtree.collapse fcard-single-1 normalize1.simps(1))
      then have max-deg (Node (Dtree.root t) {|(t2, e2)|}) = 1
        using t2-def(1) dtree.collapse[of t] by simp
      then have dom-children (Node (Dtree.root t) (sucs t)) T
        using R.dom-contr sub-t t2-def 1.prems(3) by simp
      then have dom-children t T using dtree.exhaust-sel by simp
      then have dom-children (Node r1 {|(t,e)|}) T
        using R.dom-children-child-self t2-def by simp
    then show ?thesis using dom-children-normalize1 ‹max-deg t = 1› T.wf-lverts
eq by auto
    qed
  next
    case uneq: False
    show ?thesis
    proof(cases rank (rev (Dtree.root t)) < rank (rev r1))
      case True
     then have normalize1 (Node r1 {|(t,e)|}) = Node (r1@Dtree.root t) (sucs t)
by simp
       then obtain t2 where t2-def: t2 ∈ fst ‘ fset (sucs t) is-subtree (Node r
{|(t1,e1)|}) t2
        using uneq 1.prems(1) by fastforce
      then have is-subtree t2 t using subtree-if-suc by blast
      then have is-subtree (Node r {|(t1,e1)|}) (Node r1 {|(t,e)|})
        using subtree-trans subtree-if-child t2-def(2) by auto
      then show ?thesis using R.dom-contr 1.prems(2,3) by blast
    next
      case False
       then have normalize1 (Node r1 {|(t,e)|}) = Node r1 {|(normalize1 t, e)|}
by simp
    then have is-subtree (Node r {|(t1,e1)|}) (normalize1 t) using uneq 1.prems(1)
by auto
     then show ?thesis using 1.IH False 1.prems(2,3) R.ranked-dtree-orig-rec by
simp
    qed
  qed
next
  case (2 xs r1)
  then have eq: normalize1 (Node r1 xs) = Node r1 ((λ(t,e). (normalize1 t,e)) |‘|
```

*xs*)
**using** *2.hyps* **by** *simp*
**interpret** *R*: *ranked-dtree-with-orig Node r1 xs* **using** *2.prems*(*4*) **by** *blast*
**have** ∀ *x*. ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs*) ≠ {|*x*|}
**using** *singleton-normalize1 2.hyps disjoint-darcs-if-wf-xs*[*OF R.wf-arcs*] **by** *auto*
**then have** *Node r* {|(*t1*,*e1*)|} ≠ *Node r1* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs*) **by** *auto*
**then obtain** *t3 e3* **where** *t3-def*:
(*t3*,*e3*) ∈ *fset xs is-subtree* (*Node r* {|(*t1*, *e1*)|}) (*normalize1 t3*)
**using** *2.prems*(*1*) *eq* **by** *auto*
**then show** *?case* **using** *2.IH 2.prems*(*2*,*3*) *R.ranked-dtree-orig-rec* **by** *simp*
**qed**

**lemma** *dom-children-normalize1-img-full*:
**assumes** *dom-children* (*Node r xs*) *T*
**and** ∀ (*t1*,*e1*) ∈ *fset xs*. *wf-dlverts t1*
**and** ∀ (*t1*,*e1*) ∈ *fset xs*. *max-deg t1* ≤ *1*
**shows** *dom-children* (*Node r* ((λ(*t1*,*e1*). (*normalize1 t1*,*e1*)) |‘| *xs*)) *T*
**proof** −
**have** ∀ (*t1*, *e1*) ∈ *fset xs*. *dom-children* (*Node r* {|(*t1*, *e1*)|}) *T*
**using** *dom-children-all-singletons*[*OF assms*(*1*)] **by** *blast*
**then have** ∀ (*t1*, *e1*) ∈ *fset xs*. *dom-children* (*Node r* {|(*normalize1 t1*, *e1*)|}) *T*
**using** *dom-children-normalize1 assms*(*2*,*3*) **by** *fast*
**then show** *?thesis*
**using** *dom-children-if-all-singletons*[*of* (λ(*t1*,*e1*). (*normalize1 t1*,*e1*)) |‘| *xs*] **by** *fastforce*
**qed**

**lemma** *children-deg1-normalize1-sub*:
(λ(*t1*,*e1*). (*normalize1 t1*,*e1*)) ‘ *children-deg1 xs*
⊆ *children-deg1* ((λ(*t1*,*e1*). (*normalize1 t1*,*e1*)) |‘| *xs*)
**using** *normalize1-mdeg-le order-trans* **by** *auto*

**lemma** *normalize1-children-deg1-sub-if-wfarcs*:
∀ (*t1*,*e1*)∈*fset xs*. *wf-darcs t1*
⟹ *children-deg1* ((λ(*t1*,*e1*). (*normalize1 t1*,*e1*)) |‘| *xs*)
⊆ (λ(*t1*,*e1*). (*normalize1 t1*,*e1*)) ‘ *children-deg1 xs*
**using** *normalize1-mdeg-eq* **by** *fastforce*

**lemma** *normalize1-children-deg1-eq-if-wfarcs*:
∀ (*t1*,*e1*)∈*fset xs*. *wf-darcs t1*
⟹ (λ(*t1*,*e1*). (*normalize1 t1*,*e1*)) ‘ *children-deg1 xs*
= *children-deg1* ((λ(*t1*,*e1*). (*normalize1 t1*,*e1*)) |‘| *xs*)
**using** *children-deg1-normalize1-sub normalize1-children-deg1-sub-if-wfarcs*
**by** (*meson subset-antisym*)

**lemma** *normalize1-children-deg1-sub-if-wflverts*:
∀ (*t1*,*e1*)∈*fset xs*. *wf-dlverts t1*
⟹ *children-deg1* ((λ(*t1*,*e1*). (*normalize1 t1*,*e1*)) |‘| *xs*)

```
    ⊆ (λ(t1,e1). (normalize1 t1,e1)) ‘ children-deg1 xs
  using normalize1-mdeg-eq′ by fastforce

lemma normalize1-children-deg1-eq-if-wflverts:
  ∀(t1,e1)∈fset xs. wf-dlverts t1
    ⟹ (λ(t1,e1). (normalize1 t1,e1)) ‘ children-deg1 xs
      = children-deg1 ((λ(t1,e1). (normalize1 t1,e1)) |‘| xs)
  using children-deg1-normalize1-sub normalize1-children-deg1-sub-if-wflverts
  by (meson subset-antisym)

lemma dom-children-normalize1-img:
  assumes dom-children (Node r (Abs-fset (children-deg1 xs))) T
      and ∀(t1,e1) ∈ fset xs. wf-dlverts t1
    shows dom-children (Node r (Abs-fset (children-deg1 ((λ(t1,e1). (normalize1
t1,e1)) |‘| xs)))) T
proof −
  have ∀(t1, e1) ∈ children-deg1 xs. dom-children (Node r {|(t1, e1)|}) T
    using dom-children-all-singletons[OF assms(1)] children-deg1-fset-id by blast
  then have ∀(t2, e2) ∈ (λ(t1,e1). (normalize1 t1,e1)) ‘ children-deg1 xs.
            dom-children (Node r {|(t2, e2)|}) T
    using dom-children-normalize1 assms(2) by fast
  then have ∀(t2, e2) ∈ children-deg1 ((λ(t1,e1). (normalize1 t1,e1)) |‘| xs).
            dom-children (Node r {|(t2, e2)|}) T
    using normalize1-children-deg1-eq-if-wflverts[of xs] assms(2) by blast
  then show ?thesis using dom-children-if-all-singletons children-deg1-fset-id
  proof −
    have ∀f as p. ∃pa. (dom-children (Node (as::′a list) f) p ∨ pa |∈| f) ∧ (¬ (case
pa of (d, b::′b) ⇒ dom-children (Node as {|(d, b)|}) p) ∨ dom-children (Node as
f) p)
      using dom-children-if-all-singletons by blast
     then obtain pp :: (('a list, 'b) Dtree.dtree × 'b) fset ⇒ 'a list ⇒ ('a, 'b)
pre-digraph ⇒ ('a list, 'b) Dtree.dtree × 'b where
      f1: ⋀as f p. (dom-children (Node as f) p ∨ pp f as p |∈| f) ∧ (¬ (case pp f as
p of (d, b) ⇒ dom-children (Node as {|(d, b)|}) p) ∨ dom-children (Node as f) p)
      by metis
    moreover
    { assume ¬ (case pp (Abs-fset (children-deg1 ((λ(d, y). (normalize1 d, y)) |‘|
xs))) r T of (d, b) ⇒ dom-children (Node r {|(d, b)|}) T)
      then have pp (Abs-fset (children-deg1 ((λ(d, y). (normalize1 d, y)) |‘| xs)))
r T ∉ children-deg1 ((λ(d, y). (normalize1 d, y)) |‘| xs)
        by (smt (z3) ‹∀(t2, e2) ∈children-deg1 ((λ(t1, e1). (normalize1 t1, e1))
|‘| xs). dom-children (Node r {|(t2, e2)|}) T›)
      then have pp (Abs-fset (children-deg1 ((λ(d, y). (normalize1 d, y)) |‘| xs)))
r T |∉| Abs-fset (children-deg1 ((λ(d, y). (normalize1 d, y)) |‘| xs))
        by (metis (no-types) children-deg1-fset-id)
      then have ?thesis
        using f1 by blast }
    ultimately show ?thesis
      by meson
```

```
  qed
qed

lemma normalize1-dom-wedge:
  ⟦is-subtree (Node r xs) (normalize1 t); fcard xs > 1⟧
    ⟹ dom-children (Node r (Abs-fset (children-deg1 xs))) T
using ranked-dtree-with-orig-axioms proof(induction t rule: normalize1.induct)
  case (1 r1 t e)
  then interpret R: ranked-dtree-with-orig Node r1 {|(t,e)|} by blast
  have sub-t: is-subtree (Node (Dtree.root t) (sucs t)) (Node r1 {|(t,e)|})
    using subtree-if-child[of t {|(t,e)|}] by simp
  show ?case
  proof(cases rank (rev (Dtree.root t)) < rank (rev r1))
    case True
    then have eq: normalize1 (Node r1 {|(t,e)|}) = Node (r1@Dtree.root t) (sucs
t) by simp
    then show ?thesis
    proof(cases Node r xs = normalize1 (Node r1 {|(t,e)|}))
      case True
      then have Node r xs = Node (r1@Dtree.root t) (sucs t) using eq by simp
        then show ?thesis using R.dom-wedge[OF sub-t] 1.prems(2) unfolding
dom-children-def by auto
    next
      case False
      then obtain t2 e2 where t2-def: (t2,e2) ∈ fset (sucs t) is-subtree (Node r
xs) t2
        using 1.prems(1) eq by auto
        then have is-subtree (Node r xs) t using subtree-if-suc subtree-trans by
fastforce
      then show ?thesis using R.dom-wedge sub-t 1.prems(2) by simp
    qed
  next
    case False
   then show ?thesis using 1 R.ranked-dtree-orig-rec by (auto simp: fcard-single-1)
  qed
next
  case (2 xs1 r1)
  then have eq: normalize1 (Node r1 xs1) = Node r1 ((λ(t,e). (normalize1 t,e))
|`| xs1)
    using 2.hyps by simp
  interpret R: ranked-dtree-with-orig Node r1 xs1 using 2.prems(3) by blast
  have ∀ x. ((λ(t,e). (normalize1 t,e)) |`| xs1) ≠ {|x|}
   using singleton-normalize1 2.hyps disjoint-darcs-if-wf-xs[OF R.wf-arcs] by auto
  then show ?case
  proof(cases Node r xs = normalize1 (Node r1 xs1))
    case True
    then have 1 < fcard xs1 using eq 2.prems(2) fcard-image-le less-le-trans by
fastforce
     then have dom-children (Node r1 (Abs-fset (children-deg1 xs1))) T using
```

*R.dom-wedge* **by** *simp*
**then show** *?thesis* **using** *dom-children-normalize1-img eq R.wf-lverts True* **by** *fastforce*
**next**
**case** *False*
**then show** *?thesis* **using** *2 R.ranked-dtree-orig-rec* **by** *fastforce*
**qed**
**qed**

**corollary** *normalize1-dom-wedge'*:
$\forall$ *r xs. is-subtree* (*Node r xs*) (*normalize1 t*) $\longrightarrow$ *fcard xs* > *1*
$\longrightarrow$ *dom-children* (*Node r* (*Abs-fset* {(*t*, *e*). (*t*, *e*) $\in$ *fset xs* $\wedge$ *max-deg t* $\leq$ *Suc 0*})) *T*
**by** (*auto simp only*: *normalize1-dom-wedge One-nat-def*[*symmetric*])

**lemma** *normalize1-verts-conform*: *v* $\in$ *dverts* (*normalize1 t*) $\Longrightarrow$ *seq-conform v*
**using** *ranked-dtree-with-orig-axioms* **proof**(*induction t rule*: *normalize1.induct*)
**case** *ind*: (*1 r t e*)
**then interpret** *R*: *ranked-dtree-with-orig Node r* {|(*t*, *e*)|} **by** *blast*
**consider** *rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*) *v* = *r*@*Dtree.root t*
| *rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*) *v* $\neq$ *r*@*Dtree.root t*
| $\neg$*rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*)
**by** *blast*
**then show** *?case*
**proof**(*cases*)
**case** *1*
**then show** *?thesis* **using** *R.contr-seq-conform* **by** *auto*
**next**
**case** *2*
**then have** *v* $\in$ *dverts* (*Node r* {|(*t*, *e*)|}) **using** *dverts-suc-subseteq ind.prems* **by** *fastforce*
**then show** *?thesis* **using** *R.verts-conform* **by** *blast*
**next**
**case** *3*
**then show** *?thesis* **using** *R.verts-conform ind R.ranked-dtree-orig-rec* **by** *auto*
**qed**
**next**
**case** (*2 xs r*)
**then interpret** *R*: *ranked-dtree-with-orig Node r xs* **by** *blast*
**show** *?case* **using** *R.verts-conform 2 R.ranked-dtree-orig-rec* **by** *auto*
**qed**

**corollary** *normalize1-verts-distinct*: *v* $\in$ *dverts* (*normalize1 t*) $\Longrightarrow$ *distinct v*
**using** *distinct-normalize1 verts-distinct* **by** *auto*

**lemma** *dom-mdeg-le1-aux*:
**assumes** *max-deg t* $\leq$ *1*
**and** *is-subtree* (*Node v* {|(*t2*, *e2*)|}) *t*
**and** *rank* (*rev* (*Dtree.root t2*)) < *rank* (*rev v*)

```
    and t1 ∈ fst ‘ fset (sucs t)
    and x ∈ dverts t1
  shows ∃r∈set (Dtree.root t) ∪ path-lverts t1 (hd x). r →T hd x
using assms ranked-dtree-with-orig-axioms proof(induction t arbitrary: t1)
 case (Node r xs)
 then interpret R: ranked-dtree-with-orig Node r xs by blast
 interpret T1: ranked-dtree-with-orig t1 using Node.prems(4) R.ranked-dtree-orig-rec
by force
 have fcard xs > 0 using Node.prems(4) fcard-seteq by fastforce
 then have fcard xs = 1 using mdeg-ge-fcard[of xs] Node.prems(1) by simp
 then obtain e1 where e1-def: xs = {|(t1,e1)|}
  using Node.prems(4) fcard-single-1-iff[of xs] by auto
 have mdeg1: max-deg (Node r xs) = 1
  using Node.prems(1) mdeg-ge-fcard[of xs] ‹fcard xs = 1› by simp
 show ?case
 proof(cases Node v {|(t2, e2)|} = Node r xs)
  case True
  then have dom-children (Node r xs) T
   using mdeg1 Node.prems(2,3) R.dom-contr-subtree by blast
  then show ?thesis unfolding dom-children-def using e1-def Node.prems(5)
by simp
 next
  case False
  then have sub-t1: is-subtree (Node v {|(t2, e2)|}) t1
   using Node.prems(2) e1-def is-subtree.simps[of Node v {|(t2, e2)|}] by force
  show ?thesis
  proof(cases x = Dtree.root t1)
   case True
    then show ?thesis using R.dom-sub-contr[OF self-subtree] Node.prems(3)
e1-def sub-t1 by auto
  next
   case False
   then obtain t3 where t3-def: t3 ∈ fst ‘ fset (sucs t1) x ∈ dverts t3
      using Node.prems(5) dverts-root-or-child[of x Dtree.root t1 sucs t1] by
fastforce
    have mdeg-t1: max-deg t1 ≤ 1 using mdeg-ge-child[of t1 e1 xs] e1-def mdeg1
by simp
   moreover have fcard (sucs t1) > 0 using t3-def fcard-seteq by fastforce
  ultimately have fcard (sucs t1) = 1 using mdeg-ge-fcard[of sucs t1 Dtree.root
t1] by simp
   then obtain e3 where e3-def: sucs t1 = {|(t3, e3)|}
    using t3-def fcard-single-1-iff[of sucs t1] by fastforce
   have ind: ∃r∈set (Dtree.root t1) ∪ path-lverts t3 (hd x). r →T hd x
   using Node.IH mdeg-t1 e1-def sub-t1 Node.prems(3) t3-def T1.ranked-dtree-with-orig-axioms
    by auto
  have hd x ∈ dlverts t3 using t3-def hd-in-lverts-if-wf T1.wf-lverts wf-dlverts-suc
by blast
   then have hd x ∉ set (Dtree.root t1)
    using t3-def dlverts-notin-root-sucs[OF T1.wf-lverts] by blast
```

```
    then have path-lverts t1 (hd x) = set (Dtree.root t1) ∪ path-lverts t3 (hd x)
      using path-lverts-simps1-sucs e3-def by fastforce
    then show ?thesis using ind by blast
   qed
  qed
qed

lemma dom-mdeg-le1:
  assumes max-deg t ≤ 1
     and is-subtree (Node v {|(t2, e2)|}) t
     and rank (rev (Dtree.root t2)) < rank (rev v)
   shows dom-children t T
  using dom-mdeg-le1-aux[OF assms] unfolding dom-children-def by blast

lemma dom-children-normalize1-preserv:
  assumes max-deg (normalize1 t1) ≤ 1 and dom-children t1 T and wf-dlverts
t1
  shows dom-children (normalize1 t1) T
using assms proof(induction t1 rule: normalize1.induct)
  case (1 r t e)
  then show ?case
  proof(cases rank (rev (Dtree.root t)) < rank (rev r))
    case True
    then show ?thesis using 1 dom-children-combine by force
  next
    case False
    then have max-deg (normalize1 t) ≤ 1
      using 1.prems(1) mdeg-ge-child[of normalize1 t e {|(normalize1 t,e)|}] by
simp
    then have max-deg t ≤ 1 using normalize1-mdeg-eq' 1.prems(3) by fastforce
   then show ?thesis using dom-children-normalize1 False 1.prems(2,3) by simp
  qed
next
  case (2 xs r)
  have max-deg (Node r xs) ≤ 1
    using normalize1-mdeg-eq'[OF 2.prems(3)] 2.prems(1) by fastforce
  then have fcard xs ≤ 1 using mdeg-ge-fcard[of xs] by simp
  then have fcard xs = 0 using fcard-single-1-iff[of xs] 2.hyps by fastforce
  then have normalize1 (Node r xs) = Node r xs using 2.hyps by simp
  then show ?case using 2.prems(2) by simp
qed

lemma dom-mdeg-le1-normalize1:
  assumes max-deg (normalize1 t) ≤ 1 and normalize1 t ≠ t
  shows dom-children (normalize1 t) T
proof −
  obtain v t2 e2 where is-subtree (Node v {|(t2, e2)|}) t rank (rev (Dtree.root
t2)) < rank (rev v)
    using contr-if-normalize1-uneq assms(2) by blast
```

**moreover have** *max-deg t* $\leq$ *1* **using** *assms*(*1*) *normalize1-mdeg-eq wf-arcs* **by** *fastforce*
**ultimately show** *?thesis*
**using** *dom-mdeg-le1 dom-children-normalize1-preserv assms*(*1*) *wf-lverts* **by** *blast*
**qed**

**lemma** *normalize-mdeg-eq*:
*wf-darcs t1*
$\Longrightarrow$ *max-deg* (*normalize t1*) = *max-deg t1* $\vee$ (*max-deg* (*normalize t1*) = *0* $\wedge$ *max-deg t1* = *1*)
**apply** (*induction t1 rule*: *normalize.induct*)
**by** (*smt* (*verit*, *ccfv-threshold*) *normalize1-mdeg-eq wf-darcs-normalize1 normalize.simps*)

**lemma** *normalize-mdeg-eq′*:
*wf-dlverts t1*
$\Longrightarrow$ *max-deg* (*normalize t1*) = *max-deg t1* $\vee$ (*max-deg* (*normalize t1*) = *0* $\wedge$ *max-deg t1* = *1*)
**apply** (*induction t1 rule*: *normalize.induct*)
**by** (*smt* (*verit*, *ccfv-threshold*) *normalize1-mdeg-eq′ wf-dlverts-normalize1 normalize.simps*)

**corollary** *mdeg-le1-normalize*:
⟦*max-deg* (*normalize t1*) $\leq$ *1*; *wf-dlverts t1*⟧ $\Longrightarrow$ *max-deg t1* $\leq$ *1*
**using** *normalize-mdeg-eq′* **by** *fastforce*

**lemma** *dom-children-normalize-preserv*:
**assumes** *max-deg* (*normalize t1*) $\leq$ *1* **and** *dom-children t1 T* **and** *wf-dlverts t1*
**shows** *dom-children* (*normalize t1*) *T*
**using** *assms* **proof**(*induction t1 rule*: *normalize.induct*)
**case** (*1 t1*)
**then show** *?case*
**proof**(*cases t1* = *normalize1 t1*)
**case** *True*
**then show** *?thesis* **using** *1.prems dom-children-normalize1-preserv* **by** *simp*
**next**
**case** *False*
**have** *max-deg t1* $\leq$ *1* **using** *mdeg-le1-normalize 1.prems*(*1*,*3*) **by** *blast*
**then have** *max-deg* (*normalize1 t1*) $\leq$ *1*
**using** *normalize1-mdeg-eq′ 1.prems*(*3*) **by** *fastforce*
**then have** *dom-children* (*normalize1 t1*) *T*
**using** *dom-children-normalize1-preserv 1.prems*(*2*,*3*) **by** *blast*
**then show** *?thesis* **using** *1 False* **by** (*simp add*: *Let-def wf-dlverts-normalize1*)
**qed**
**qed**

**lemma** *dom-mdeg-le1-normalize*:
**assumes** *max-deg* (*normalize t*) $\leq$ *1* **and** *normalize t* $\neq$ *t*

```
  shows dom-children (normalize t) T
using assms ranked-dtree-with-orig-axioms proof(induction t rule: normalize.induct)
  case (1 t)
  then interpret T: ranked-dtree-with-orig t by blast
  show ?case
    using 1 T.dom-mdeg-le1-normalize1 T.wf-lverts wf-dlverts-normalize1
   by (smt (verit) dom-children-normalize-preserv normalize.elims mdeg-le1-normalize)
qed

lemma normalize1-arc-in-dlverts:
  ⟦is-subtree (Node v ys) (normalize1 t); x ∈ set v; x →_T y⟧ ⟹ y ∈ dlverts (Node
v ys)
using ranked-dtree-with-orig-axioms proof(induction t rule: normalize1.induct)
  case ind: (1 r t e)
  then interpret R: ranked-dtree-with-orig Node r {|(t, e)|} by blast
  show ?case
  proof(cases rank (rev (Dtree.root t)) < rank (rev r))
    case True
   then have eq: normalize1 (Node r {|(t, e)|}) = Node (r@Dtree.root t) (sucs t)
by simp
    then show ?thesis
    proof(cases Node v ys = Node (r@Dtree.root t) (sucs t))
      case True
     then consider x ∈ set r | x ∈ set (Dtree.root t) using ind.prems(2) by auto
      then show ?thesis
      proof(cases)
        case 1
        then have y ∈ dlverts (Node r {|(t, e)|})
          using R.arc-in-dlverts ind.prems(3) by fastforce
       then show ?thesis using eq normalize1-dlverts-eq[of Node r {|(t, e)|}] True
by simp
      next
        case 2
        then have y ∈ dlverts t
          using R.arc-in-dlverts[of Dtree.root t sucs t] ind.prems(3)
            subtree-if-child[of t {|(t, e)|}] by simp
       then show ?thesis using eq normalize1-dlverts-eq[of Node r {|(t, e)|}] True
by simp
      qed
    next
      case False
      then obtain t2 where t2-def: t2 ∈ fst ‘ fset (sucs t) is-subtree (Node v ys)
t2
        using ind.prems(1) eq by force
      then have is-subtree (Node v ys) (Node r {|(t, e)|})
        using subtree-trans[OF t2-def(2)] subtree-if-suc by auto
      then show ?thesis using R.arc-in-dlverts ind.prems(2,3) by blast
    qed
  next
```

**case** *nocontr*: *False*
**then show** *?thesis*
**proof**(*cases Node v ys = Node r* {|(*normalize1 t*, *e*)|})
**case** *True*
**then have** *y* ∈ *dlverts* (*Node r* {|(*t*, *e*)|})
**using** *R.arc-in-dlverts ind.prems*(*2*,*3*) **by** *fastforce*
**then show** *?thesis* **using** *nocontr True* **by** *simp*
**next**
**case** *False*
**then have** *is-subtree* (*Node v ys*) (*normalize1 t*) **using** *ind.prems*(*1*) *nocontr*
**by** *auto*
**then show** *?thesis* **using** *ind.IH*[*OF nocontr*] *ind.prems*(*2*,*3*) *R.ranked-dtree-orig-rec*
**by** *simp*
**qed**
**qed**
**next**
**case** (*2 xs r*)
**then interpret** *R*: *ranked-dtree-with-orig Node r xs* **by** *blast*
**have** *eq*: *normalize1* (*Node r xs*) = *Node r* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs*)
**using** *2.hyps* **by** *simp*
**show** *?case*
**proof**(*cases Node v ys* = *normalize1* (*Node r xs*))
**case** *True*
**then have** *y* ∈ *dlverts* (*Node r xs*) **using** *R.arc-in-dlverts 2.hyps 2.prems*(*2*,*3*)
**by** *simp*
**then show** *?thesis* **using** *True* **by** *simp*
**next**
**case** *False*
**then obtain** *t2 e2* **where** *t2-def*: (*t2*,*e2*) ∈ *fset xs is-subtree* (*Node v ys*)
(*normalize1 t2*)
**using** *2.hyps 2.prems*(*1*) **by** *auto*
**then show** *?thesis* **using** *2.IH 2.prems*(*2*,*3*) *R.ranked-dtree-orig-rec* **by** *simp*
**qed**
**qed**

**lemma** *normalize1-arc-in-dlverts′*:
∀ *r xs*. *is-subtree* (*Node r xs*) (*normalize1 t*) ⟶ (∀ *x*. *x* ∈ *set r*
⟶ (∀ *y*. *x* →$_T$ *y* ⟶ *y* ∈ *set r* ∨ (∃ *x*∈*fset xs*. *y* ∈ *dlverts* (*fst x*))))
**using** *normalize1-arc-in-dlverts* **by** *simp*

**theorem** *ranked-dtree-orig-normalize1*: *ranked-dtree-with-orig* (*normalize1 t*) *rank*
*cost cmp T root*
**by** (*simp add*: *ranked-dtree-with-orig-def ranked-dtree-with-orig-axioms-def asi-rank*
*normalize1-dom-contr normalize1-dom-mdeg-gt1 normalize1-dom-sub-contr*
*normalize1-dom-wedge′ directed-tree-axioms normalize1-arc-in-dlverts′*
*ranked-dtree-normalize1 normalize1-verts-conform normalize1-verts-distinct*)

**theorem** *ranked-dtree-orig-normalize*: *ranked-dtree-with-orig* (*normalize t*) *rank*
*cost cmp T root*

**using** *ranked-dtree-with-orig-axioms* **proof**(*induction t rule*: *normalize.induct*)
**case** (*1 t*)
**then interpret** *T*: *ranked-dtree-with-orig t* **by** *blast*
**show** *?case* **using** *1.IH T.ranked-dtree-orig-normalize1* **by**(*auto simp*: *Let-def*)
**qed**

### 10.3.2 Merging preserves Arc Invariants

**interpretation** *Comm*: *comp-fun-commute merge-f r xs* **by** (*rule merge-commute*)

**lemma** *path-lverts-supset-z*:
⟦*list-dtree* (*Node r xs*); ∀ *t1* ∈ *fst* ‘ *fset xs*. *a* ∉ *dlverts t1*⟧
⟹ *path-lverts-list z a* ⊆ *path-lverts-list* (*ffold* (*merge-f r xs*) *z xs*) *a*
**proof**(*induction xs*)
**case** (*insert x xs*)
**interpret** *Comm*: *comp-fun-commute merge-f r* (*finsert x xs*) **by** (*rule merge-commute*)
**define** *f* **where** *f* = *merge-f r* (*finsert x xs*)
**define** *f′* **where** *f′* = *merge-f r xs*
**let** *?merge* = *Sorting-Algorithms.merge cmp′*
**have** *0*: *list-dtree* (*Node r xs*) **using** *list-dtree-subset insert.prems*(*1*) **by** *blast*
**show** *?case*
**proof**(*cases ffold f z* (*finsert x xs*) = *ffold f′ z xs*)
**case** *True*
**then show** *?thesis* **using** *insert.IH 0 insert.prems*(*2*) *f-def f′-def* **by** *auto*
**next**
**case** *False*
**obtain** *t2 e2* **where** *t2-def*[*simp*]: *x* = (*t2*,*e2*) **by** *fastforce*
**have** *1*: ∀ *v*∈*fst* ‘ *set* (*dtree-to-list* (*Node r* {|(*t2*, *e2*)|})). *a* ∉ *set v*
**using** *insert.prems*(*2*) *dtree-to-list-x-in-dlverts* **by** *auto*
**have** *xs* |⊆| *finsert x xs* **by** *blast*
**then have** *f-xs*: *ffold f z xs* = *ffold f′ z xs*
**using** *merge-ffold-supset insert.prems*(*1*) *f-def f′-def* **by** *presburger*
**have** *ffold f z* (*finsert x xs*) = *f x* (*ffold f z xs*)
**using** *Comm.ffold-finsert*[*OF insert.hyps*] *f-def* **by** *blast*
**then have** *2*: *ffold f z* (*finsert x xs*) = *f x* (*ffold f′ z xs*) **using** *f-xs* **by** *argo*
**then have** *f x* (*ffold f′ z xs*) ≠ *ffold f′ z xs* **using** *False f-def f′-def* **by** *argo*
**then have** *f* (*t2*,*e2*) (*ffold f′ z xs*)
= *?merge* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|})) (*ffold f′ z xs*)
**using** *merge-f-merge-if-not-snd t2-def f-def* **by** *blast*
**then have** *ffold f z* (*finsert x xs*)
= *?merge* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|})) (*ffold f′ z xs*)
**using** *2 t2-def* **by** *argo*
**then have** *path-lverts-list* (*ffold f′ z xs*) *a* ⊆ *path-lverts-list* (*ffold f z* (*finsert x xs*)) *a*
**using** *path-lverts-list-merge-supset-ys-notin*[*OF 1*] **by** *presburger*
**then show** *?thesis* **using** *insert.IH 0 insert.prems*(*2*) *f-def f′-def* **by** *auto*
**qed**
**qed** (*simp add*: *ffold.rep-eq*)

**lemma** *path-lverts-merge-ffold-sup*:
⟦*list-dtree* (*Node r xs*); *t1* ∈ *fst* ‘ *fset xs*; *a* ∈ *dlverts t1*⟧
⟹ *path-lverts t1 a* ⊆ *path-lverts-list* (*ffold* (*merge-f r xs*) [] *xs*) *a*
**proof**(*induction xs*)
**case** (*insert x xs*)
**interpret** *Comm*: *comp-fun-commute merge-f r* (*finsert x xs*) **by** (*rule merge-commute*)
**define** *f* **where** *f* = *merge-f r* (*finsert x xs*)
**define** *f′* **where** *f′* = *merge-f r xs*
**let** *?merge* = *Sorting-Algorithms.merge cmp′*
**have** *0*: *list-dtree* (*Node r xs*) **using** *list-dtree-subset insert.prems*(*1*) **by** *blast*
**obtain** *t2 e2* **where** *t2-def*[*simp*]: *x* = (*t2*,*e2*) **by** *fastforce*
**have** (*t2*, *e2*) ∈ *fset* (*finsert x xs*) **by** *simp*
**moreover have** (*t2*, *e2*) ∉ *fset xs* **using** *insert.hyps* **by** *fastforce*
**ultimately have** *xs-val*:
(∀(*v*,*e*) ∈ *set* (*ffold f′* [] *xs*). *set v* ∩ *dlverts t2* = {} ∧ *v* ≠ [] ∧ *e* ∉ *darcs t2* ∪ {*e2*})
**using** *merge-ffold-empty-inter-preserv′*[*OF insert.prems*(*1*) *empty-list-valid-merge*] *f′-def*
**by** *blast*
**have** *ffold f* [] (*finsert x xs*) = *f x* (*ffold f* [] *xs*)
**using** *Comm.ffold-finsert*[*OF insert.hyps*] *f-def* **by** *blast*
**also have** ... = *f x* (*ffold f′* [] *xs*)
**using** *merge-ffold-supset*[*of xs finsert x xs r* []] *insert.prems*(*1*) *f-def f′-def* **by** *fastforce*
**finally have** *ffold f* [] (*finsert x xs*) = *?merge* (*dtree-to-list* (*Node r* {|*x*|})) (*ffold f′* [] *xs*)
**using** *merge-f-merge-if-conds xs-val insert.prems f-def* **by** *simp*
**then have** *merge*: *ffold f* [] (*finsert x xs*)
= *?merge* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|})) (*ffold f′*[] *xs*)
**using** *t2-def* **by** *blast*
**show** *?case*
**proof**(*cases t1* = *t2*)
**case** *True*
**then have** ∀ *v*∈*fst* ‘ *set* (*ffold f′* [] *xs*). *a* ∉ *set v*
**using** *insert.prems*(*3*) *xs-val* **by** *fastforce*
**then have** *path-lverts-list* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|})) *a*
⊆ *path-lverts-list* (*ffold f* [] (*finsert x xs*)) *a*
**using** *merge path-lverts-list-merge-supset-xs-notin* **by** *fastforce*
**then show** *?thesis* **using** *True f-def path-lverts-to-list-eq* **by** *force*
**next**
**case** *False*
**then have** *a* ∉ *dlverts t2* **using** *insert.prems list-dtree.wf-lverts* **by** *fastforce*
**then have** *1*: ∀ *v*∈*fst* ‘ *set* (*dtree-to-list* (*Node r* {|(*t2*, *e2*)|})). *a* ∉ *set v*
**using** *dtree-to-list-x-in-dlverts* **by** *fast*
**have** *path-lverts t1 a* ⊆ *path-lverts-list* (*ffold f′* [] *xs*) *a*
**using** *insert.IH*[*OF 0*] *insert.prems*(*2*,*3*) *False f′-def* **by** *simp*
**then show** *?thesis* **using** *f-def merge path-lverts-list-merge-supset-ys-notin*[*OF 1*] **by** *auto*
**qed**

**qed**(*simp*)

**lemma** *path-lverts-merge-sup-aux*:
**assumes** *list-dtree* (*Node r xs*) **and** *t1* ∈ *fst* ‘ *fset xs* **and** *a* ∈ *dlverts t1*
**and** *ffold* (*merge-f r xs*) [] *xs* = (*v1*, *e1*) # *ys*
**shows** *path-lverts t1 a* ⊆ *path-lverts* (*dtree-from-list v1 ys*) *a*
**proof** −
**have** *xs* ≠ {||} **using** *assms*(*2*) **by** *auto*
**have** *path-lverts t1 a* ⊆ *path-lverts-list* (*ffold* (*merge-f r xs*) [] *xs*) *a*
**using** *path-lverts-merge-ffold-sup*[*OF assms*(*1*−*3*)] **.**
**then show** *?thesis* **using** *path-lverts-from-list-eq assms*(*4*) **by** *fastforce*
**qed**

**lemma** *path-lverts-merge-sup*:
**assumes** *list-dtree* (*Node r xs*) **and** *t1* ∈ *fst* ‘ *fset xs* **and** *a* ∈ *dlverts t1*
**shows** ∃ *t2 e2*. *merge* (*Node r xs*) = *Node r* {|(*t2*,*e2*)|}
∧ *path-lverts t1 a* ⊆ *path-lverts t2 a*
**proof** −
**have** *xs* ≠ {||} **using** *assms*(*2*) **by** *auto*
**then obtain** *t2 e2* **where** *t2-def*: *merge* (*Node r xs*) = *Node r* {|(*t2*,*e2*)|}
**using** *merge-singleton*[*OF assms*(*1*)] **by** *blast*
**obtain** *y ys* **where** *y-def*: *ffold* (*merge-f r xs*) [] *xs* = *y* # *ys*
**using** *merge-ffold-nempty*[*OF assms*(*1*) ‹*xs* ≠ {||}›] *list.exhaust-sel* **by** *blast*
**obtain** *v1 e1* **where** *y* = (*v1*,*e1*) **by** *fastforce*
**then show** *?thesis* **using** *merge-xs path-lverts-merge-sup-aux*[*OF assms*] *t2-def*
*y-def* **by** *fastforce*
**qed**

**lemma** *path-lverts-merge-sup-sucs*:
**assumes** *list-dtree t0* **and** *t1* ∈ *fst* ‘ *fset* (*sucs t0*) **and** *a* ∈ *dlverts t1*
**shows** ∃ *t2 e2*. *merge t0* = *Node* (*Dtree.root t0*) {|(*t2*,*e2*)|}
∧ *path-lverts t1 a* ⊆ *path-lverts t2 a*
**using** *path-lverts-merge-sup*[*of Dtree.root t0 sucs t0*] *assms* **by** *simp*

**lemma** *merge-dom-children-aux*:
**assumes** *list-dtree t0*
**and** ∀ *x*∈*dverts t1*. ∃ *v* ∈ *set* (*Dtree.root t0*) ∪ *path-lverts t1* (*hd x*). *v* →$_T$ *hd*
*x*
**and** *t1* ∈ *fst* ‘ *fset* (*sucs t0*)
**and** *wf-dlverts t1*
**and** *x* ∈ *dverts t1*
**shows** ∃!*t2* ∈ *fst* ‘ *fset* (*sucs* (*merge t0*)).
∃ *v* ∈ *set* (*Dtree.root* (*merge t0*)) ∪ *path-lverts t2* (*hd x*). *v* →$_T$ (*hd x*)
**proof** −
**have** *hd x* ∈ *dlverts t1* **using** *assms*(*4*,*5*) **by** (*simp add*: *hd-in-lverts-if-wf*)
**then obtain** *t2 e2* **where** *t2-def*:
*merge t0* = *Node* (*Dtree.root t0*) {|(*t2*,*e2*)|} *path-lverts t1* (*hd x*) ⊆ *path-lverts*
*t2* (*hd x*)
**using** *path-lverts-merge-sup-sucs*[*OF assms*(*1*,*3*)] **by** *blast*

**then show** *?thesis* **using** *assms*(*2*,*5*) **by** *force*
**qed**

**lemma** *merge-dom-children-aux'*:
**assumes** *dom-children t0 T*
**and** ∀ *t1* ∈ *fst* ‘ *fset* (*sucs t0*). *wf-dlverts t1*
**and** *t2* ∈ *fst* ‘ *fset* (*sucs* (*merge t0*))
**and** *x* ∈ *dverts t2*
**shows** ∃ *v*∈*set* (*Dtree.root* (*merge t0*)) ∪ *path-lverts t2* (*hd x*). *v* →$_T$ *hd x*
**proof** −
**have** *disj*: *list-dtree t0*
**using** *assms*(*3*) *merge-empty-if-nwf-sucs*[*of t0*] **by** *fastforce*
**obtain** *t1* **where** *t1-def*: *t1* ∈ *fst* ‘ *fset* (*sucs t0*) *x* ∈ *dverts t1*
**using** *verts-child-if-merge-child*[*OF assms*(*3*,*4*)] **by** *blast*
**then have** *0*: ∀ *x*∈*dverts t1*. ∃ *v*∈*set* (*Dtree.root t0*) ∪ *path-lverts t1* (*hd x*). *v* →$_T$ *hd x*
**using** *assms*(*1*) **unfolding** *dom-children-def* **by** *blast*
**then have** *wf-dlverts t1* **using** *t1-def*(*1*) *assms*(*2*) **by** *blast*
**then obtain** *t3* **where** *t3-def*: *t3* ∈ *fst* ‘ *fset* (*sucs* (*merge t0*))
(∃ *v*∈*set* (*Dtree.root* (*merge t0*)) ∪ *path-lverts t3* (*hd x*). *v* →$_T$ *hd x*)
**using** *merge-dom-children-aux*[*OF disj 0*] *t1-def* **by** *blast*
**then have** *t3* = *t2* **using** *assms*(*3*) *merge-single-root1-sucs* **by** *fastforce*
**then show** *?thesis* **using** *t3-def*(*2*) **by** *blast*
**qed**

**lemma** *merge-dom-children-sucs*:
**assumes** *dom-children t0 T* **and** ∀ *t1* ∈ *fst* ‘ *fset* (*sucs t0*). *wf-dlverts t1*
**shows** *dom-children* (*merge t0*) *T*
**using** *merge-dom-children-aux'*[*OF assms*] *dom-children-def* **by** *fast*

**lemma** *merge-dom-children*:
⟦*dom-children* (*Node r xs*) *T*; ∀ *t1* ∈ *fst* ‘ *fset xs*. *wf-dlverts t1*⟧
⟹ *dom-children* (*merge* (*Node r xs*)) *T*
**using** *merge-dom-children-sucs* **by** *auto*

**lemma** *merge-dom-children-if-ndisjoint*:
¬*list-dtree* (*Node r xs*) ⟹ *dom-children* (*merge* (*Node r xs*)) *T*
**using** *merge-empty-if-nwf* **unfolding** *dom-children-def* **by** *simp*

**lemma** *merge-subtree-fcard-le1*: *is-subtree* (*Node r xs*) (*merge t1*) ⟹ *fcard xs* ≤ *1*
**using** *merge-mdeg-le1-sub le-trans mdeg-ge-fcard* **by** *fast*

**lemma** *merge-dom-mdeg-gt1*:
⟦*is-subtree* (*Node r xs*) (*merge t2*); *t1* ∈ *fst* ‘ *fset xs*; *max-deg* (*Node r xs*) > *1*⟧
⟹ ∃ *v* ∈ *set r*. *v* →$_T$ *hd* (*Dtree.root t1*)
**using** *merge-mdeg-le1-sub* **by** *fastforce*

**lemma** *merge-root-if-contr*:

$[\![\bigwedge r1\ t2\ e2.\ is\text{-}subtree\ (Node\ r1\ \{|(t2,e2)|\})\ t1 \Longrightarrow rank\ (rev\ r1) \leq rank\ (rev\ (Dtree.root\ t2));$
$is\text{-}subtree\ (Node\ v\ \{|(t2,e2)|\})\ (merge\ t1);\ rank\ (rev\ (Dtree.root\ t2)) < rank\ (rev\ v)]\!]$
$\Longrightarrow Node\ v\ \{|(t2,e2)|\} = merge\ t1$
**using** *merge-strict-subtree-nocontr-sucs2*[*of t1 v*] *strict-subtree-def* **by** *fastforce*

**lemma** *merge-new-contr-fcard-gt1*:
**assumes** $\bigwedge r1\ t2\ e2.\ is\text{-}subtree\ (Node\ r1\ \{|(t2,e2)|\})\ t1 \Longrightarrow rank\ (rev\ r1) \leq rank\ (rev\ (Dtree.root\ t2))$
**and** $Node\ v\ \{|(t2,e2)|\} = (merge\ t1)$
**and** $rank\ (rev\ (Dtree.root\ t2)) < rank\ (rev\ v)$
**shows** $fcard\ (sucs\ t1) > 1$
**proof** −
**have** *t-v*: $Dtree.root\ t1 = v$ **using** *assms*(2) *dtree.sel*(1)[*of v* $\{|(t2,e2)|\}$] **by** *simp*
**have** $\forall\, t2\ e2.\ Node\ v\ \{|(t2,e2)|\} \neq t1$
**using** *assms merge-root-child-eq self-subtree less-le-not-le* **by** *metis*
**then have** $\forall\, x.\ sucs\ t1 \neq \{|x|\}$ **using** *t-v dtree.collapse*[*of t1*] **by** *force*
**moreover have** $sucs\ t1 \neq \{||\}$ **using** *assms*(2) *merge-empty-sucs* **by** *force*
**ultimately show** *?thesis* **using** *fcard-single-1-iff*[*of sucs t1*] *fcard-0-eq*[*of sucs t1*] **by** *force*
**qed**

**lemma** *merge-dom-sub-contr-if-nocontr*:
**assumes** $\bigwedge r1\ t2\ e2.\ is\text{-}subtree\ (Node\ r1\ \{|(t2,e2)|\})\ t \Longrightarrow rank\ (rev\ r1) \leq rank\ (rev\ (Dtree.root\ t2))$
**and** $is\text{-}subtree\ (Node\ r\ xs)\ (merge\ t)$
**and** $t1 \in fst\ `\ fset\ xs$
**and** $\exists\, v\ t2\ e2.\ is\text{-}subtree\ (Node\ v\ \{|(t2,e2)|\})\ (Node\ r\ xs)$
$\wedge\ rank\ (rev\ (Dtree.root\ t2)) < rank\ (rev\ v)$
**shows** $\exists\, v \in set\ r.\ v \rightarrow_T hd\ (Dtree.root\ t1)$
**proof** −
**obtain** *v t2 e2* **where** *t2-def*:
$is\text{-}subtree\ (Node\ v\ \{|(t2,e2)|\})\ (Node\ r\ xs)\ rank\ (rev\ (Dtree.root\ t2)) < rank\ (rev\ v)$
**using** *assms*(4) **by** *blast*
**then have** $is\text{-}subtree\ (Node\ v\ \{|(t2,e2)|\})\ (merge\ t)$ **using** *assms*(2) *subtree-trans* **by** *blast*
**then have** *eq*: $Node\ v\ \{|(t2,e2)|\} = merge\ t$ **using** *merge-root-if-contr assms*(1) *t2-def*(2) **by** *blast*
**then have** *t-v*: $Dtree.root\ t = v$ **using** *dtree.sel*(1)[*of v* $\{|(t2,e2)|\}$] **by** *simp*
**have** *eq2*: $Node\ v\ \{|(t2,e2)|\} = Node\ r\ xs$
**using** *eq assms*(2) *t2-def*(1) *subtree-antisym*[*of Node v* $\{|(t2,\ e2)|\}$] **by** *simp*
**have** $fcard\ (sucs\ t) > 1$ **using** *merge-new-contr-fcard-gt1*[*OF assms*(1) *eq t2-def*(2)] **by** *simp*
**then have** *mdeg*: $max\text{-}deg\ t > 1$ **using** *mdeg-ge-fcard*[*of sucs t Dtree.root t*] **by** *simp*
**have** *sub*: $is\text{-}subtree\ (Node\ (Dtree.root\ t)\ (sucs\ t))\ t$ **using** *self-subtree*[*of t*] **by** *simp*

**obtain** *e1* **where** *e1-def*: (*t1*, *e1*)∈*fset* (*sucs* (*merge t*))
**using** *assms*(*3*) *eq eq2 dtree.sel*(*2*)[*of r xs*] **by** *force*
**then obtain** *t3* **where** *t3-def*: (*t3*, *e1*)∈*fset* (*sucs t*) *Dtree.root t3* = *Dtree.root t1*
**using** *merge-child-in-orig*[*OF e1-def*] **by** *blast*
**then have** ∃ *v*∈*set* (*Dtree.root t*). *v* →$_T$ *hd* (*Dtree.root t1*) **using** *dom-mdeg-gt1 sub mdeg* **by** *fastforce*
**then show** *?thesis* **using** *t-v eq2* **by** *blast*
**qed**

**lemma** *merge-dom-contr-if-nocontr-mdeg-le1*:
**assumes** ⋀*r1 t2 e2*. *is-subtree* (*Node r1* {|(*t2*,*e2*)|}) *t* ⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*Dtree.root t2*))
**and** *is-subtree* (*Node r* {|(*t1*,*e1*)|}) (*merge t*)
**and** *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*)
**and** ∀ *t* ∈ *fst* ‘ *fset* (*sucs t*). *max-deg t* ≤ *1*
**shows** *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T*
**proof** −
**have** *eq*: *Node r* {|(*t1*,*e1*)|} = *merge t* **using** *merge-root-if-contr*[*OF assms*(*1*−*3*)]
.
**have** *0*: ∀ *t1*∈*fst* ‘ *fset* (*sucs t*). *wf-dlverts t1* **using** *wf-lverts wf-dlverts-suc* **by** *auto*
**have** *fcard* (*sucs t*) > *1* **using** *merge-new-contr-fcard-gt1*[*OF assms*(*1*) *eq assms*(*3*)] **by** *simp*
**then have** *dom-children t T* **using** *dom-wedge-full*[*of Dtree.root t*] *assms*(*4*) *self-subtree* **by** *force*
**then show** *?thesis* **using** *merge-dom-children-sucs 0 eq* **by** *simp*
**qed**

**lemma** *merge-dom-wedge*:
⟦*is-subtree* (*Node r xs*) (*merge t1*); *fcard xs* > *1*; ∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*⟧
⟹ *dom-children* (*Node r xs*) *T*
**using** *merge-subtree-fcard-le1* **by** *fastforce*

### 10.3.3 Merge1 preserves Arc Invariants

**lemma** *merge1-dom-mdeg-gt1*:
**assumes** *is-subtree* (*Node r xs*) (*merge1 t*) **and** *t1* ∈ *fst* ‘ *fset xs* **and** *max-deg* (*Node r xs*) > *1*
**shows** ∃ *v* ∈ *set r*. *v* →$_T$ *hd* (*Dtree.root t1*)
**proof** −
**obtain** *ys* **where** *ys-def*: *merge1* (*Node r ys*) = *Node r xs is-subtree* (*Node r ys*) *t*
**using** *merge1-subtree-if-mdeg-gt1*[*OF assms*(*1*,*3*)] **by** *blast*
**then obtain** *t3* **where** *t3-def*: *t3* ∈ *fst* ‘ *fset ys Dtree.root t3* = *Dtree.root t1*
**using** *assms*(*2*) *merge1-child-in-orig* **by** *fastforce*
**have** *max-deg* (*Node r ys*) > *1* **using** *merge1-mdeg-le*[*of Node r ys*] *ys-def*(*1*) *assms*(*3*) **by** *simp*

**then show** *?thesis* **using** *dom-mdeg-gt1*[*OF ys-def*(*2*) *t3-def*(*1*)] *t3-def* **by** *simp*
**qed**

**lemma** *max-deg1-gt-1-if-new-contr*:
**assumes** ⋀*r1 t2 e2*. *is-subtree* (*Node r1* {|(*t2*,*e2*)|}) *t0* ⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*Dtree.root t2*))
**and** *is-subtree* (*Node r* {|(*t1*,*e1*)|}) (*merge1 t0*)
**and** *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*)
**shows** *max-deg t0* > *1*
**using** *assms merge1-mdeg-gt1-if-uneq* **by** *force*

**lemma** *merge1-subtree-if-new-contr*:
**assumes** ⋀*r1 t2 e2*. *is-subtree* (*Node r1* {|(*t2*,*e2*)|}) *t0* ⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*Dtree.root t2*))
**and** *is-subtree* (*Node r xs*) (*merge1 t0*)
**and** *is-subtree* (*Node v* {|(*t1*,*e1*)|}) (*Node r xs*)
**and** *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev v*)
**shows** ∃ *ys*. *is-subtree* (*Node r ys*) *t0* ∧ *merge1* (*Node r ys*) = *Node r xs*
**using** *assms* **proof**(*induction t0*)
**case** (*Node r' ys*)
**then consider** *fcard ys* > *1* (∀ *t* ∈ *fst* ‘ *fset ys*. *max-deg t* ≤ *1*)
| ¬(*fcard ys* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset ys*. *max-deg t* ≤ *1*)) *Node r xs* = *merge1* (*Node r' ys*)
| ¬(*fcard ys* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset ys*. *max-deg t* ≤ *1*)) *Node r xs* ≠ *merge1* (*Node r' ys*)
**by** *blast*
**then show** *?case*
**proof**(*cases*)
**case** *1*
**then have** *is-subtree* (*Node v* {|(*t1*, *e1*)|}) (*merge* (*Node r' ys*))
**using** *subtree-trans*[*OF Node.prems*(*3*,*2*)] **by** *force*
**then have** *Node v* {|(*t1*, *e1*)|} = *merge* (*Node r' ys*)
**using** *merge-root-if-contr Node.prems*(*1*,*4*) **by** *blast*
**then have** *Node r xs* = *merge1* (*Node r' ys*)
**using** *Node.prems*(*2*,*3*) *1 subtree-eq-if-trans-eq1* **by** *fastforce*
**then show** *?thesis* **using** *1 dtree.sel*(*1*)[*of r xs*] **by** *auto*
**next**
**case** *2*
**then have** *r* = *r'* **using** *dtree.sel*(*1*)[*of r xs*] **by** *force*
**then show** *?thesis* **using** *2*(*2*) **by** *auto*
**next**
**case** *3*
**then have** *merge1* (*Node r' ys*) = *Node r'* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *ys*) **by** *auto*
**then obtain** *t2 e2* **where** *t2-def*: (*t2*,*e2*) ∈ *fset ys is-subtree* (*Node r xs*) (*merge1 t2*)
**using** *Node.prems*(*2*) *3*(*2*) **by** *auto*
**then have** *subt2*: *is-subtree t2* (*Node r' ys*) **using** *subtree-if-child*
**by** (*metis fstI image-eqI*)

**then have** $\bigwedge r1\ t3\ e3.\ is\text{-}subtree\ (Node\ r1\ \{|(t3,\ e3)|\})\ t2$
$\Longrightarrow rank\ (rev\ r1) \leq rank\ (rev\ (Dtree.root\ t3))$
**using** *Node.prems*(*1*) *subtree-trans* **by** *blast*
**then obtain** *ys′* **where** *ys-def*: *is-subtree* (*Node r ys′*) *t2 merge1* (*Node r ys′*) = *Node r xs*
**using** *Node.IH*[*OF t2-def*(*1*)] *Node.prems*(*3*,*4*) *t2-def*(*2*) **by** *auto*
**then show** *?thesis* **using** *subtree-trans subt2* **by** *blast*
**qed**
**qed**

**lemma** *merge1-dom-sub-contr*:
**assumes** $\bigwedge r1\ t2\ e2.\ is\text{-}subtree\ (Node\ r1\ \{|(t2,e2)|\})\ t \Longrightarrow rank\ (rev\ r1) \leq rank\ (rev\ (Dtree.root\ t2))$
**and** *is-subtree* (*Node r xs*) (*merge1 t*)
**and** *t1* $\in$ *fst* ‘ *fset xs*
**and** $\exists\, v\ t2\ e2.\ is\text{-}subtree\ (Node\ v\ \{|(t2,e2)|\})\ (Node\ r\ xs) \wedge rank\ (rev\ (Dtree.root\ t2)) < rank\ (rev\ v)$
**shows** $\exists\, v \in set\ r.\ v \rightarrow_T hd\ (Dtree.root\ t1)$
**proof** −
**obtain** *ys* **where** *ys-def*: *is-subtree* (*Node r ys*) *t merge1* (*Node r ys*) = *Node r xs*
**using** *merge1-subtree-if-new-contr assms*(*1*,*2*,*4*) **by** *blast*
**then interpret** *R*: *ranked-dtree-with-orig Node r ys* **using** *ranked-dtree-orig-subtree* **by** *blast*
**obtain** *v t2 e2* **where** *v-def*:
*is-subtree* (*Node v* {|(*t2*,*e2*)|}) (*Node r xs*) *rank* (*rev* (*Dtree.root t2*)) < *rank* (*rev v*)
**using** *assms*(*4*) **by** *blast*
**then have** *is-subtree* (*Node v* {|(*t2*,*e2*)|}) (*merge1* (*Node r ys*)) **using** *ys-def* **by** *simp*
**then have** *mdeg-gt1*: *max-deg* (*Node r ys*) > *1*
**using** *max-deg1-gt-1-if-new-contr assms*(*1*) *v-def*(*2*) *subtree-trans ys-def*(*1*) **by** *blast*
**obtain** *t3* **where** *t3-def*: *t3* $\in$ *fst* ‘ *fset ys Dtree.root t3* = *Dtree.root t1*
**using** *ys-def*(*2*) *assms*(*3*) *merge1-child-in-orig* **by** *fastforce*
**then show** *?thesis* **using** *R.dom-mdeg-gt1*[*OF self-subtree*] *mdeg-gt1* **by** *fastforce*
**qed**

**lemma** *merge1-merge-point-if-new-contr*:
**assumes** $\bigwedge r1\ t2\ e2.\ is\text{-}subtree\ (Node\ r1\ \{|(t2,e2)|\})\ t0 \Longrightarrow rank\ (rev\ r1) \leq rank\ (rev\ (Dtree.root\ t2))$
**and** *wf-darcs t0*
**and** *is-subtree* (*Node r* {|(*t1*,*e1*)|}) (*merge1 t0*)
**and** *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*)
**shows** $\exists\, ys.\ is\text{-}subtree\ (Node\ r\ ys)\ t0 \wedge fcard\ ys > 1 \wedge (\forall\, t \in fst\ ‘\ fset\ ys.\ max\text{-}deg\ t \leq 1)$
$\wedge$ *merge1* (*Node r ys*) = *Node r* {|(*t1*,*e1*)|}
**using** *assms* **proof**(*induction t0*)
**case** (*Node v xs*)

```
  then consider fcard xs > 1 (∀ t ∈ fst ‘ fset xs. max-deg t ≤ 1)
    | fcard xs ≤ 1 | fcard xs > 1 ¬(∀ t ∈ fst ‘ fset xs. max-deg t ≤ 1)
    by linarith
  then show ?case
  proof(cases)
    case 1
   then have is-subtree (Node r {|(t1, e1)|}) (merge (Node v xs)) using Node.prems(3)
by simp
    then have Node r {|(t1, e1)|} = merge (Node v xs)
      using merge-root-if-contr Node.prems(1,4) by blast
    then show ?thesis using 1 dtree.sel(1)[of r {|(t1, e1)|}] by auto
  next
    case 2
   then have merge1 (Node v xs) = Node v ((λ(t,e). (merge1 t,e)) |‘| xs) by auto
    then have xs ≠ {||} using Node.prems(3) by force
    then have fcard xs = 1 using 2 le-Suc-eq by auto
    then obtain t2 e2 where t2-def: xs = {|(t2,e2)|} using fcard-single-1-iff[of
xs] by fast
   then have Node r {|(t1, e1)|} ≠ merge1 (Node v {|(t2,e2)|}) using Node.prems(1,4)
2 by force
    then have is-subtree (Node r {|(t1, e1)|}) (merge1 t2) using Node.prems(3)
t2-def 2 by auto
    moreover have ⋀r1 t3 e3. is-subtree (Node r1 {|(t3, e3)|}) t2
                  ⟹ rank (rev r1) ≤ rank (rev (Dtree.root t3))
      using Node.prems(1) t2-def by fastforce
     ultimately show ?thesis using Node.IH[of (t2,e2)] Node.prems(2,4) t2-def
by fastforce
  next
    case 3
    then have fcard ((λ(t,e). (merge1 t,e)) |‘| xs) > 1
      using fcard-merge1-img-if-disjoint disjoint-darcs-if-wf-xs[OF Node.prems(2)]
by simp
    then have Node r {|(t1,e1)|} ≠ merge1 (Node v xs)
      using fcard-single-1-iff[of (λ(t,e). (merge1 t,e)) |‘| xs] 3(2) by auto
    moreover have merge1 (Node v xs) = Node v ((λ(t,e). (merge1 t,e)) |‘| xs)
using 3(2) by auto
    ultimately obtain t2 e2 where t2-def:
       (t2,e2) ∈ fset xs is-subtree (Node r {|(t1, e1)|}) (merge1 t2)
      using Node.prems(3) by auto
    then have is-subtree t2 (Node v xs) using subtree-if-child
      by (metis fst-conv image-eqI)
    then have ⋀r1 t3 e3. is-subtree (Node r1 {|(t3, e3)|}) t2
                  ⟹ rank (rev r1) ≤ rank (rev (Dtree.root t3))
      using Node.prems(1) subtree-trans by blast
    then obtain ys where ys-def: is-subtree (Node r ys) t2 1 < fcard ys
       (∀ t∈fst ‘ fset ys. max-deg t ≤ 1) merge1 (Node r ys) = Node r {|(t1, e1)|}
      using Node.IH[OF t2-def(1)] Node.prems(2,4) t2-def by fastforce
    then show ?thesis using t2-def(1) by auto
  qed
```

**qed**

**lemma** *merge1-dom-contr*:
**assumes** $\bigwedge r1\ t2\ e2.$ *is-subtree* (*Node r1* {|(*t2*,*e2*)|}) *t* $\Longrightarrow$ *rank* (*rev r1*) $\leq$ *rank* (*rev* (*Dtree.root t2*))
**and** *is-subtree* (*Node r* {|(*t1*,*e1*)|}) (*merge1 t*)
**and** *rank* (*rev* (*Dtree.root t1*)) $<$ *rank* (*rev r*)
**and** *max-deg* (*Node r* {|(*t1*,*e1*)|}) = *1*
**shows** *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T*
**proof** −
**obtain** *ys* **where** *ys-def*: *is-subtree* (*Node r ys*) *t fcard ys* > *1*
$\forall\, t{\in}fst\ `\ fset\ ys.\ max\text{-}deg\ t \leq 1\ merge1\ (Node\ r\ ys) = Node\ r\ \{|(t1,e1)|\}$
**using** *merge1-merge-point-if-new-contr wf-arcs assms*(*1*−*3*) **by** *blast*
**have** $\forall\, t1{\in}fst\ `\ fset\ ys.$ *wf-dlverts t1*
**using** *ys-def*(*1*) *list-dtree.wf-lverts list-dtree-sub* **by** *fastforce*
**then show** *?thesis* **using** *merge-dom-children-sucs*[*OF dom-wedge-full*] *ys-def* **by** *fastforce*
**qed**

**lemma** *merge1-dom-children-merge-sub-aux*:
**assumes** *merge1 t* = *t2*
**and** *is-subtree* (*Node r′ xs′*) *t*
**and** *fcard xs′* > *1*
**and** $(\forall\, t{\in}fst\ `\ fset\ xs'.\ max\text{-}deg\ t \leq 1)$
**and** *max-deg t2* $\leq$ *1*
**and** *x* $\in$ *dverts t2*
**and** *x* $\neq$ *Dtree.root t2*
**shows** $\exists\, v \in$ *path-lverts t2* (*hd x*). *v* $\rightarrow_T$ *hd x*
**using** *assms ranked-dtree-with-orig-axioms* **proof**(*induction t arbitrary*: *t2*)
**case** (*Node r xs*)
**then interpret** *R*: *ranked-dtree-with-orig Node r xs* **by** *blast*
**obtain** *t1 e1* **where** *t1-def*: (*t1*,*e1*) $\in$ *fset* (*sucs t2*) *x* $\in$ *dverts t1*
**by** (*metis Node.prems*(*6*,*7*) *fsts.simps dtree.sel dtree.set-cases*(*1*) *fst-conv surj-pair*)
**then have** *t2-sucs*: *sucs t2* = {|(*t1*,*e1*)|}
**using** *Node.prems*(*5*) *empty-iff-mdeg-0*[*of Dtree.root t2 sucs t2*]
*mdeg-1-singleton*[*of Dtree.root t2 sucs t2*] **by** *auto*
**have** *wf-t2*: *wf-dlverts t2* **using** *Node.prems*(*1*) *R.wf-dlverts-merge1* **by** *blast*
**then have** *wf-dlverts t1* **using** *t1-def*(*1*) *wf-dlverts-suc* **by** *fastforce*
**then have** *hd x* $\in$ *dlverts t1* **using** *t1-def*(*2*) *hd-in-lverts-if-wf* **by** *blast*
**then have** *hd x* $\notin$ *set* (*Dtree.root t2*) **using** *dlverts-notin-root-sucs*[*OF wf-t2*] *t1-def*(*1*) **by** *fastforce*
**then have** *path-t2*: *path-lverts t2* (*hd x*) = *set* (*Dtree.root t2*) $\cup$ *path-lverts t1* (*hd x*)
**using** *path-lverts-simps1-sucs t2-sucs* **by** *fastforce*
**show** *?case*
**proof**(*cases Node r xs* = *Node r′ xs′*)
**case** *True*
**then have** *merge* (*Node r′ xs′*) = *t2* **using** *Node.prems*(*1*,*3*,*4*) **by** *simp*
**then have** *dom-children t2 T*

**using** *R.dom-wedge-full*[*OF Node.prems*(2−4)] *merge-dom-children R.wf-lverts True* **by** *fastforce*
**then have** ∃ *v*∈*set* (*Dtree.root t2*) ∪ *path-lverts t1* (*hd x*). *v* →$_T$ *hd x*
**using** *t1-def* **unfolding** *dom-children-def* **by** *auto*
**then show** *?thesis* **using** *path-t2* **by** *blast*
**next**
**case** *False*
**then have** ¬(*fcard xs* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs. max-deg t* ≤ *1*))
**using** *Node.prems*(3,4) *child-mdeg-gt1-if-sub-fcard-gt1*[*OF Node.prems*(2)] **by** *force*
**then have** *eq*: *merge1* (*Node r xs*) = *Node r* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs*) **by** *auto*
**then obtain** *t3 e3* **where** *t3-def*: (*t3*,*e3*) ∈ *fset xs is-subtree* (*Node r′ xs′*) *t3*
**using** *Node.prems*(2) *False* **by** *auto*
**have** *fcard* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs*) = *1*
**using** *Node.prems*(1) *eq t2-sucs fcard-single-1* **by** *fastforce*
**then have** *fcard xs* = *1*
**using** *fcard-merge1-img-if-disjoint disjoint-darcs-if-wf-xs*[*OF R.wf-arcs*] **by** *simp*
**then have** *xs* = {|(*t3*,*e3*)|} **using** *fcard-single-1-iff*[*of xs*] *t3-def*(1) **by** *auto*
**then have** *t13*: *merge1 t3* = *t1* **using** *t2-sucs eq Node.prems*(1) **by** *force*
**then have** *mdegt3*: *max-deg t1* ≤ *1*
**using** *Node.prems*(5) *mdeg-ge-child*[*of t1 e1 sucs t2 Dtree.root t2*] *t2-sucs* **by** *fastforce*
**have** *mdeg-gt1*: *max-deg* (*Node r xs*) > *1*
**using** *mdeg-ge-fcard*[*of xs′ r′*] *Node.prems*(2,3) *mdeg-ge-sub*[*of Node r′ xs′ Node r xs*]
**by** *simp*
**show** *?thesis*
**proof**(*cases x* = *Dtree.root t1*)
**case** *True*
**then have** ∃ *v*∈*set r. v* →$_T$ *hd x*
**using** *R.dom-mdeg-gt1*[*of r xs*] *t3-def*(1) *mdeg-gt1 t13* **by** *fastforce*
**then show** *?thesis* **using** *path-t2 Node.prems*(1) **by** *auto*
**next**
**case** *False*
**then have** ∃ *v*∈*path-lverts t1* (*hd x*). *v* →$_T$ *hd x*
**using** *Node.IH t1-def*(2) *t3-def t13 assms*(3,4) *mdegt3 R.ranked-dtree-orig-rec* **by** *simp*
**then show** *?thesis* **using** *path-t2* **by** *blast*
**qed**
**qed**
**qed**

**lemma** *merge1-dom-children-fcard-gt1-aux*:
**assumes** *dom-children* (*Node r* (*Abs-fset* (*children-deg1 ys*))) *T*
**and** *is-subtree* (*Node r ys*) *t*
**and** *merge1* (*Node r ys*) = *Node r xs*
**and** *fcard xs* > *1*

```
      and max-deg t2 ≤ 1
      and t2 ∈ fst ‘ fset xs
      and x ∈ dverts t2
    shows ∃ v∈set r ∪ path-lverts t2 (hd x). v →_T hd x
proof −
  obtain t1 where t1-def: t1 ∈ fst ‘ fset ys merge1 t1 = t2
    using merge1-elem-in-img-if-fcard-gt1[OF assms(3,4)] assms(6) by fastforce
  then have x-t: x ∈ dverts t1 using merge1-dverts-sub assms(7) by blast
  show ?thesis
  proof(cases max-deg t1 ≤ 1)
    case True
    then have t1 ∈ fst ‘ fset (sucs (Node r (Abs-fset (children-deg1 ys))))
      using t1-def(1) children-deg1-fset-id by force
    then have ∃ v∈set r ∪ path-lverts t1 (hd x). v →_T hd x
      using assms(1) x-t unfolding dom-children-def by auto
     then show ?thesis using t1-def(2) merge1-mdeg-gt1-if-uneq[of t1] True by
force
  next
    case False
    then obtain r' xs' where r'-def:
        is-subtree (Node r' xs') t1 1 < fcard xs' (∀ t∈fst ‘ fset xs'. max-deg t ≤ 1)
      using merge1-wedge-if-uneq[of t1] assms(5) t1-def(2) by fastforce
    interpret R: ranked-dtree-with-orig Node r ys using ranked-dtree-orig-subtree
assms(2) .
    interpret T: ranked-dtree-with-orig t1 using R.ranked-dtree-orig-rec t1-def(1)
by force
    have max-deg (Node r ys) > 1
      using assms(3,4) merge1-fcard-le[of r ys] mdeg-ge-fcard[of ys] by simp
    show ?thesis
    proof (cases x = Dtree.root t2)
      case True
      have max-deg (Node r ys) > 1
        using assms(3,4) merge1-fcard-le[of r ys] mdeg-ge-fcard[of ys] by simp
    then show ?thesis using dom-mdeg-gt1[OF assms(2) t1-def(1)] True t1-def(2)
by auto
    next
      case False
      then show ?thesis
      using T.merge1-dom-children-merge-sub-aux[OF t1-def(2) r'-def assms(5,7)]
by blast
    qed
  qed
qed

lemma merge1-dom-children-fcard-gt1:
  assumes dom-children (Node r (Abs-fset (children-deg1 ys))) T
      and is-subtree (Node r ys) t
      and merge1 (Node r ys) = Node r xs
      and fcard xs > 1
```

**shows** *dom-children* (*Node r* (*Abs-fset* (*children-deg1 xs*))) *T*
**unfolding** *dom-children-def*
**using** *merge1-dom-children-fcard-gt1-aux*[*OF assms*] *children-deg1-fset-id*[*of xs*]
**by** *fastforce*

**lemma** *merge1-dom-wedge*:
**assumes** *is-subtree* (*Node r xs*) (*merge1 t*) **and** *fcard xs* > *1*
**shows** *dom-children* (*Node r* (*Abs-fset* (*children-deg1 xs*))) *T*
**proof** −
**obtain** *ys* **where** *ys-def*:
*merge1* (*Node r ys*) = *Node r xs is-subtree* (*Node r ys*) *t fcard xs* ≤ *fcard ys*
**using** *merge1-subtree-if-fcard-gt1*[*OF assms*] **by** *blast*
**have** *dom-children* (*Node r* (*Abs-fset* (*children-deg1 ys*))) *T*
**using** *dom-wedge ys-def*(*2*,*3*) *assms*(*2*) **by** *simp*
**then show** *?thesis* **using** *merge1-dom-children-fcard-gt1 ys-def*(*2*,*1*) *assms*(*2*)
**by** *blast*
**qed**

**corollary** *merge1-dom-wedge′*:
∀ *r xs*. *is-subtree* (*Node r xs*) (*merge1 t*) ⟶ *fcard xs* > *1*
⟶ *dom-children* (*Node r* (*Abs-fset* {(*t*, *e*). (*t*, *e*) ∈ *fset xs* ∧ *max-deg t* ≤ *Suc 0*})) *T*
**by** (*auto simp only*: *merge1-dom-wedge One-nat-def*[*symmetric*])

**corollary** *merge1-verts-conform*: *v* ∈ *dverts* (*merge1 t*) ⟹ *seq-conform v*
**by** (*simp add*: *verts-conform*)

**corollary** *merge1-verts-distinct*: ⟦*v* ∈ *dverts* (*merge1 t*)⟧ ⟹ *distinct v*
**using** *distinct-merge1 verts-distinct* **by** *auto*

**lemma** *merge1-mdeg-le1-wedge-if-fcard-gt1*:
**assumes** *max-deg* (*merge1 t1*) ≤ *1*
**and** *wf-darcs t1*
**and** *is-subtree* (*Node v ys*) *t1*
**and** *fcard ys* > *1*
**shows** (∀ *t* ∈ *fst* ‘ *fset ys*. *max-deg t* ≤ *1*)
**using** *assms* **proof**(*induction t1 rule*: *merge1.induct*)
**case** (*1 r xs*)
**then show** *?case*
**proof**(*cases fcard xs* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*))
**case** *True*
**then have** *Node v ys* = *Node r xs*
**using** *1.prems*(*3*,*4*) *mdeg-ge-sub mdeg-ge-fcard*[*of ys*] **by** *fastforce*
**then show** *?thesis* **using** *True* **by** *simp*
**next**
**case** *False*
**then have** *eq*: *merge1* (*Node r xs*) = *Node r* ((λ(*t*, *e*). (*merge1 t*, *e*)) |‘| *xs*) **by** *auto*
**have** *fcard* ((λ(*t*, *e*). (*merge1 t*, *e*)) |‘| *xs*) = *fcard xs*

**using** *fcard-merge1-img-if-disjoint disjoint-darcs-if-wf-xs*[*OF 1.prems*(*2*)] **by** *simp*
**then have** *fcard xs* ≤ *1*
**by** (*metis 1.prems*(*1*) *False merge1.simps num-leaves-1-if-mdeg-1 num-leaves-ge-card*)
**then have** *Node v ys* ≠ *Node r xs* **using** *1.prems*(*4*) **by** *auto*
**then obtain** *t2 e2* **where** *t2-def*: (*t2*,*e2*) ∈ *fset xs is-subtree* (*Node v ys*) *t2*
**using** *1.prems*(*3*) **by** *auto*
**then have** *max-deg* (*merge1 t2*) ≤ *1*
**using** *1.prems*(*1*) *False eq*
*mdeg-ge-child*[*of merge1 t2 e2* (λ(*t*, *e*). (*merge1 t*, *e*)) |‘| *xs*]
**by** *fastforce*
**then show** *?thesis* **using** *1.IH*[*OF False t2-def*(*1*) *refl*] *t2-def 1.prems*(*2*,*4*) **by** *fastforce*
**qed**
**qed**

**lemma** *dom-mdeg-le1-merge1-aux*:
**assumes** *max-deg* (*merge1 t*) ≤ *1*
**and** *merge1 t* ≠ *t*
**and** *t1* ∈ *fst* ‘ *fset* (*sucs* (*merge1 t*))
**and** *x* ∈ *dverts t1*
**shows** ∃ *r*∈*set* (*Dtree.root* (*merge1 t*)) ∪ *path-lverts t1* (*hd x*). *r* →$_T$ *hd x*
**using** *assms ranked-dtree-with-orig-axioms* **proof**(*induction t arbitrary*: *t1 rule*: *merge1.induct*)
**case** (*1 r xs*)
**then interpret** *R*: *ranked-dtree-with-orig Node r xs* **by** *blast*
**show** *?case*
**proof**(*cases fcard xs* > *1*)
**case** *True*
**then have** *0*: (∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*)
**using** *merge1-mdeg-le1-wedge-if-fcard-gt1*[*OF 1.prems*(*1*) *R.wf-arcs*] **by** *auto*
**then have** *dom-children* (*merge* (*Node r xs*)) *T*
**using** *True merge-dom-children-sucs R.dom-wedge-full R.wf-lverts self-subtree wf-dlverts-suc*
**by** *fast*
**then show** *?thesis* **unfolding** *dom-children-def* **using** *1.prems*(*3*,*4*) *0 True* **by** *auto*
**next**
**case** *False*
**then have** *rec*: ¬(*fcard xs* > *1* ∧ (∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*)) **by** *simp*
**then have** *eq*: *merge1* (*Node r xs*) = *Node r* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs*) **by** *auto*
**obtain** *t2 e2* **where** *t2-def*: *xs* = {|(*t2*,*e2*)|} *merge1 t2* = *t1*
**using** *1.prems*(*3*) *False singleton-if-fcard-le1-elem*[*of xs*] **by** *fastforce*
**show** *?thesis*
**proof**(*cases x* = *Dtree.root t1*)
**case** *True*
**have** *max-deg* (*Node r xs*) > *1* **using** *merge1-mdeg-gt1-if-uneq 1.prems*(*2*) **by** *blast*

**then show** *?thesis* **using** *True R.dom-mdeg-gt1*[*OF self-subtree*] *t2-def* **by** *auto*
**next**
**case** *False*
**then obtain** *t3* **where** *t3-def*: *t3* ∈ *fst* ‘ *fset* (*sucs* (*merge1 t2*)) *x* ∈ *dverts t3*
**using** *1.prems*(*4*) *t2-def*(*2*) *dverts-root-or-suc* **by** *fastforce*
**have** *mdeg1*: *max-deg* (*merge1 t2*) ≤ *1*
**using** *1.prems*(*1*) *mdeg-ge-child*[*of t1 e2* (λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs*] *eq t2-def*
**by** *simp*
**then have** *0*: ∃ *r*∈*set* (*Dtree.root* (*merge1 t2*)) ∪ *path-lverts t3* (*hd x*). *r* →$_T$ *hd x*
**using** *1.IH rec mdeg1 t3-def 1.prems*(*2*) *eq t2-def R.ranked-dtree-orig-rec* **by** *auto*
**obtain** *e3* **where** *e3-def*: *sucs t1* = {|(*t3*, *e3*)|}
**using** *t3-def singleton-if-mdeg-le1-elem-suc mdeg1 t2-def*(*2*) **by** *fastforce*
**have** *wf-dlverts t1* **using** *wf-dlverts-suc 1.prems*(*3*) *R.wf-dlverts-merge1* **by** *blast*
**then have** *hd x* ∈ *dlverts t3*
**using** *t3-def*(*2*) *1.prems*(*4*) *list-in-verts-iff-lverts hd-in-set*[*of x*] *empty-notin-wf-dlverts*
**by** *fast*
**then have** *hd x* ∉ *set* (*Dtree.root t1*)
**using** *t3-def*(*1*) *dlverts-notin-root-sucs*[*OF* ‹*wf-dlverts t1*›] *t2-def*(*2*) **by** *blast*
**then show** *?thesis* **using** *0 path-lverts-simps1-sucs*[*of hd x t1*] *e3-def t2-def*(*2*) **by** *blast*
**qed**
**qed**
**qed**

**lemma** *dom-mdeg-le1-merge1*:
⟦*max-deg* (*merge1 t*) ≤ *1*; *merge1 t* ≠ *t*⟧ ⟹ *dom-children* (*merge1 t*) *T*
**unfolding** *dom-children-def* **using** *dom-mdeg-le1-merge1-aux* **by** *blast*

**lemma** *merge1-arc-in-dlverts*:
⟦*is-subtree* (*Node r xs*) (*merge1 t*); *x* ∈ *set r*; *x* →$_T$ *y*⟧ ⟹ *y* ∈ *dlverts* (*Node r xs*)
**using** *merge1-subtree-dlverts-supset arc-in-dlverts* **by** *blast*

**theorem** *merge1-ranked-dtree-orig*:
**assumes** ⋀*r1 t2 e2*. *is-subtree* (*Node r1* {|(*t2*,*e2*)|}) *t* ⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*Dtree.root t2*))
**shows** *ranked-dtree-with-orig* (*merge1 t*) *rank cost cmp T root*
**using** *assms merge1-arc-in-dlverts*
**unfolding** *ranked-dtree-with-orig-def ranked-dtree-with-orig-axioms-def*
**by**(*simp add*: *directed-tree-axioms ranked-dtree-merge1 merge1-verts-distinct merge1-verts-conform merge1-dom-mdeg-gt1 merge1-dom-contr merge1-dom-sub-contr merge1-dom-wedge′ asi-rank*)

**theorem** *merge1-normalize-ranked-dtree-orig*:
*ranked-dtree-with-orig* (*merge1* (*normalize t*)) *rank cost cmp T root*
**using** *ranked-dtree-with-orig.merge1-ranked-dtree-orig*[*OF ranked-dtree-orig-normalize*]
**by** (*simp add*: *normalize-sorted-ranks*)

**theorem** *ikkbz-sub-ranked-dtree-orig*: *ranked-dtree-with-orig* (*ikkbz-sub t*) *rank cost cmp T root*
**using** *ranked-dtree-with-orig-axioms* **proof**(*induction t rule*: *ikkbz-sub.induct*)
**case** (*1 t*)
**then show** *?case*
**proof**(*cases max-deg t* ≤ *1*)
**case** *True*
**then show** *?thesis* **using** *1.prems* **by** *auto*
**next**
**case** *False*
**then show** *?thesis*
**by** (*metis 1 ranked-dtree-with-orig.merge1-normalize-ranked-dtree-orig ikkbz-sub.simps*)
**qed**
**qed**

## 10.4 Optimality of IKKBZ-Sub result constrained to Invariants

**lemma** *dtree-size-skip-decr*[*termination-simp*]: *size* (*Node r* (*sucs t1*)) < *size* (*Node v* {|(*t1*,*e1*)|})
**using** *dtree-size-eq-root*[*of Dtree.root t1 sucs t1*] **by** *auto*

**lemma** *dtree-size-skip-decr1*: *size* (*Node* (*r* @ *Dtree.root t1*) (*sucs t1*)) < *size* (*Node r* {|(*t1*,*e1*)|})
**using** *dtree-size-skip-decr* **by** *auto*

**function** *normalize-full* :: ('*a list*,'*b*) *dtree* ⇒ ('*a list*,'*b*) *dtree* **where**
*normalize-full* (*Node r* {|(*t1*,*e1*)|}) = *normalize-full* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
| ∀ *x*. *xs* ≠ {|*x*|} ⟹ *normalize-full* (*Node r xs*) = *Node r xs*
**using** *dtree-to-list.cases* **by** *blast+*
**termination using** *dtree-size-skip-decr termination in-measure wf-measure* **by** *metis*

### 10.4.1 Result fulfills the requirements

**lemma** *ikkbz-sub-eq-if-mdeg-le1*: *max-deg t1* ≤ *1* ⟹ *ikkbz-sub t1* = *t1*
**by** *simp*

**lemma** *ikkbz-sub-eq-iff-mdeg-le1*: *max-deg t1* ≤ *1* ⟷ *ikkbz-sub t1* = *t1*
**using** *ikkbz-sub-mdeg-le1*[*of t1*] **by** *fastforce*

**lemma** *dom-mdeg-le1-ikkbz-sub*: *ikkbz-sub t* ≠ *t* ⟹ *dom-children* (*ikkbz-sub t*) *T*
**using** *ranked-dtree-with-orig-axioms* **proof**(*induction t rule*: *ikkbz-sub.induct*)

```
  case (1 t)
  then interpret T: ranked-dtree-with-orig t by simp
  interpret NT: ranked-dtree-with-orig normalize t
    using T.ranked-dtree-orig-normalize by blast
  interpret MT: ranked-dtree-with-orig merge1 (normalize t)
    using T.merge1-normalize-ranked-dtree-orig by blast
  show ?case
  proof(cases max-deg t ≤ 1)
    case True
    then show ?thesis using 1.prems by auto
  next
    case False
    then show ?thesis
    proof(cases max-deg (merge1 (normalize t)) ≤ 1)
      case True
      then show ?thesis
      using NT.dom-mdeg-le1-merge1 T.dom-mdeg-le1-normalize T.list-dtree-axioms
False
        by force
    next
      case False
      then have ikkbz-sub (merge1 (normalize t)) ≠ (merge1 (normalize t))
        using ikkbz-sub-mdeg-le1[of merge1 (normalize t)] by force
      then show ?thesis using 1 MT.ranked-dtree-with-orig-axioms by auto
    qed
  qed
qed

lemma combine-denormalize-eq:
  denormalize (Node r {|(t1,e1)|}) = denormalize (Node (r@Dtree.root t1) (sucs
t1))
  by (induction t1 rule: denormalize.induct) auto

lemma normalize1-denormalize-eq: wf-dlverts t1 ⟹ denormalize (normalize1 t1)
= denormalize t1
proof(induction t1 rule: normalize1.induct)
  case (1 r t e)
  then show ?case using combine-denormalize-eq[of r t] by simp
next
  case (2 xs r)
  then show ?case
    using fcard-single-1-iff[of (λ(t,e). (normalize1 t,e)) |`| xs] fcard-single-1-iff[of
xs]
    by (auto simp: fcard-normalize-img-if-wf-dlverts)
qed

lemma normalize1-denormalize-eq': wf-darcs t1 ⟹ denormalize (normalize1 t1)
= denormalize t1
proof(induction t1 rule: normalize1.induct)
```

**case** (*1 r t e*)
**then show** *?case* **using** *combine-denormalize-eq*[*of r t*] **by** (*auto simp*: *wf-darcs-iff-darcs'*)
**next**
**case** (*2 xs r*)
**then show** *?case*
**using** *fcard-single-1-iff*[*of* (λ(*t*,*e*). (*normalize1 t*,*e*)) |`| *xs*] *fcard-single-1-iff*[*of xs*]
**by** (*auto simp*: *fcard-normalize-img-if-disjoint wf-darcs-iff-darcs'*)
**qed**

**lemma** *normalize-denormalize-eq*: *wf-dlverts t1* ⟹ *denormalize* (*normalize t1*) = *denormalize t1*
**apply** (*induction t1 rule*: *normalize.induct*)
**by** (*smt* (*verit*) *normalize1-denormalize-eq normalize.simps wf-dlverts-normalize1*)

**lemma** *normalize-denormalize-eq'*: *wf-darcs t1* ⟹ *denormalize* (*normalize t1*) = *denormalize t1*
**apply** (*induction t1 rule*: *normalize.induct*)
**by** (*smt* (*verit*) *normalize1-denormalize-eq' normalize.simps wf-darcs-normalize1*)

**lemma** *normalize-full-denormalize-eq*[*simp*]: *denormalize* (*normalize-full t1*) = *denormalize t1*
**proof**(*induction t1 rule*: *normalize-full.induct*)
**case** (*1 r t e*)
**then show** *?case* **using** *combine-denormalize-eq*[*of r t*] **by** *simp*
**qed**(*simp*)

**lemma** *combine-dlverts-eq*: *dlverts* (*Node r* {|(*t1*,*e1*)|}) = *dlverts* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
**using** *dlverts.simps*[*of Dtree.root t1 sucs t1*] **by** *auto*

**lemma** *normalize-full-dlverts-eq*[*simp*]: *dlverts* (*normalize-full t1*) = *dlverts t1*
**using** *combine-dlverts-eq* **by**(*induction t1 rule*: *normalize-full.induct*) *fastforce*+

**lemma** *combine-darcs-sub*: *darcs* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*)) ⊆ *darcs* (*Node r* {|(*t1*,*e1*)|})
**using** *dtree.set*(*2*)[*of Dtree.root t1 sucs t1*] **by** *auto*

**lemma** *normalize-full-darcs-sub*: *darcs* (*normalize-full t1*) ⊆ *darcs t1*
**using** *combine-darcs-sub* **by**(*induction t1 rule*: *normalize-full.induct*) *fastforce*+

**lemma** *combine-nempty-if-wf-dlverts*: *wf-dlverts* (*Node r* {|(*t1*,*e1*)|}) ⟹ *r* @ *Dtree.root t1* ≠ []
**by** *simp*

**lemma** *combine-empty-inter-if-wf-dlverts*:
**assumes** *wf-dlverts* (*Node r* {|(*t1*,*e1*)|})
**shows** ∀(*x*, *e1*)∈*fset* (*sucs t1*). *set* (*r* @ *Dtree.root t1*) ∩ *dlverts x* = {} ∧ *wf-dlverts x*

**proof** −
**have** $\forall (x,\ e1) \in fset\ (sucs\ t1).\ set\ r \cap dlverts\ x = \{\}$ **using** *suc-in-dlverts assms*
**by** *fastforce*
**then show** *?thesis* **using** *wf-dlverts.simps*[*of Dtree.root t1 sucs t1*] *assms* **by** *auto*
**qed**

**lemma** *combine-disjoint-if-wf-dlverts*:
*wf-dlverts* (*Node r* {|(*t1*,*e1*)|}) ⟹ *disjoint-dlverts* (*sucs t1*)
**using** *wf-dlverts.simps*[*of Dtree.root t1 sucs t1*] **by** *simp*

**lemma** *combine-wf-dlverts*:
*wf-dlverts* (*Node r* {|(*t1*,*e1*)|}) ⟹ *wf-dlverts* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
**using** *combine-empty-inter-if-wf-dlverts*[*of r t1*] *wf-dlverts.simps*[*of Dtree.root t1 sucs t1*]
**by** *force*

**lemma** *combine-distinct*:
**assumes** ∀ *v* ∈ *dverts* (*Node r* {|(*t1*,*e1*)|}). *distinct v*
**and** *wf-dlverts* (*Node r* {|(*t1*,*e1*)|})
**and** *v* ∈ *dverts* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
**shows** *distinct v*
**proof**(*cases v* = *r* @ *Dtree.root t1*)
**case** *True*
**have** (*Dtree.root t1*) ∈ *dverts t1* **by** (*simp add*: *dtree.set-sel*(*1*))
**moreover from** *this* **have** *set r* ∩ *set* (*Dtree.root t1*) = {}
**using** *assms*(*2*) *lverts-if-in-verts* **by** *fastforce*
**ultimately show** *?thesis* **using** *True assms*(*1*) **by** *simp*
**next**
**case** *False*
**then show** *?thesis* **using** *assms*(*1*,*3*) *dverts-suc-subseteq* **by** *fastforce*
**qed**

**lemma** *normalize-full-wfdlverts*: *wf-dlverts t1* ⟹ *wf-dlverts* (*normalize-full t1*)
**proof**(*induction t1 rule*: *normalize-full.induct*)
**case** (*1 r t1 e1*)
**then show** *?case* **using** *combine-wf-dlverts*[*of r t1*] **by** *simp*
**qed**(*simp*)

**corollary** *normalize-full-wfdverts*: *wf-dlverts t1* ⟹ *wf-dverts* (*normalize-full t1*)
**using** *normalize-full-wfdlverts* **by** (*simp add*: *wf-dverts-if-wf-dlverts*)

**lemma** *combine-wf-arcs*: *wf-darcs* (*Node r* {|(*t1*,*e1*)|}) ⟹ *wf-darcs* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
**using** *wf-darcs'.simps*[*of Dtree.root t1 sucs t1*] **by** (*simp add*: *wf-darcs-iff-darcs'*)

**lemma** *normalize-full-wfdarcs*: *wf-darcs t1* ⟹ *wf-darcs* (*normalize-full t1*)
**using** *combine-wf-arcs* **by**(*induction t1 rule*: *normalize-full.induct*) *fastforce*+

**lemma** *normalize-full-dom-preserv*: *dom-children t1 T* $\Longrightarrow$ *dom-children* (*normalize-full t1*) *T*
**by** (*induction t1 rule*: *normalize-full.induct*) (*auto simp*: *dom-children-combine*)

**lemma** *combine-forward*:
**assumes** *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T*
**and** $\forall$ *v* $\in$ *dverts* (*Node r* {|(*t1*,*e1*)|}). *forward v*
**and** *wf-dlverts* (*Node r* {|(*t1*,*e1*)|})
**and** *v* $\in$ *dverts* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
**shows** *forward v*
**proof**(*cases v* = *r* @ *Dtree.root t1*)
**case** *True*
**have** *0*: (*Dtree.root t1*) $\in$ *dverts t1* **by** (*simp add*: *dtree.set-sel*(*1*))
**then have** *fwd-t1*: *forward* (*Dtree.root t1*) **using** *assms*(*2*) **by** *simp*
**moreover have** *set r* $\cap$ *set* (*Dtree.root t1*) = {} **using** *assms*(*3*) *0 lverts-if-in-verts* **by** *fastforce*
**moreover have** $\exists$ *x*$\in$*set r*. $\exists$ *y*$\in$*set* (*Dtree.root t1*). *x* $\rightarrow_T$ *y*
**using** *assms*(*1*,*3*) *root-arc-if-dom-wfdlverts* **by** *fastforce*
**ultimately have** $\exists$ *x*$\in$*set r*. *x* $\rightarrow_T$ *hd* (*Dtree.root t1*) **using** *forward-arc-to-head* **by** *blast*
**moreover have** *fwd-r*: *forward r* **using** *assms*(*2*) **by** *simp*
**ultimately show** *?thesis* **using** *forward-app fwd-t1 True* **by** *simp*
**next**
**case** *False*
**then show** *?thesis* **using** *assms*(*2*,*4*) *dverts-suc-subseteq* **by** *fastforce*
**qed**

**lemma** *normalize-full-forward*:
⟦*dom-children t1 T*; $\forall$ *v* $\in$ *dverts t1*. *forward v*; *wf-dlverts t1*⟧
$\Longrightarrow$ $\forall$ *v* $\in$ *dverts* (*normalize-full t1*). *forward v*
**proof**(*induction t1 rule*: *normalize-full.induct*)
**case** (*1 r t e*)
**have** $\forall$ *v* $\in$ *dverts* (*Node* (*r*@*Dtree.root t*) (*sucs t*)). *forward v*
**using** *combine-forward*[*OF 1.prems*(*1*,*2*,*3*)] **by** *blast*
**moreover have** *dom-children* (*Node* (*r*@*Dtree.root t*) (*sucs t*)) *T*
**using** *dom-children-combine 1.prems*(*1*) **by** *simp*
**ultimately show** *?case* **using** *1.IH 1.prems*(*3*) *combine-wf-dlverts*[*of r t e*] **by** *fastforce*
**qed**(*auto*)

**lemma** *normalize-full-max-deg0*: *max-deg t1* $\leq$ *1* $\Longrightarrow$ *max-deg* (*normalize-full t1*) = *0*
**proof**(*induction t1 rule*: *normalize-full.induct*)
**case** (*1 r t e*)
**then show** *?case* **using** *mdeg-child-sucs-le* **by** (*fastforce dest*: *order-trans*)
**next**
**case** (*2 xs r*)
**then show** *?case* **using** *empty-fset-if-mdeg-le1-not-single* **by** *auto*
**qed**

**lemma** *normalize-full-mdeg-eq*: *max-deg t1 > 1 ⟹ max-deg* (*normalize-full t1*) = *max-deg t1*
**proof**(*induction t1 rule*: *normalize-full.induct*)
**case** (*1 r t e*)
**then show** *?case* **using** *mdeg-child-sucs-eq-if-gt1* **by** *force*
**qed**(*auto*)

**lemma** *normalize-full-empty-sucs*: *max-deg t1 ≤ 1 ⟹* ∃ *r*. *normalize-full t1* = *Node r* {||}
**proof**(*induction t1 rule*: *normalize-full.induct*)
**case** (*1 r t e*)
**then show** *?case* **using** *mdeg-child-sucs-le* **by** (*fastforce dest*: *order-trans*)
**next**
**case** (*2 xs r*)
**then show** *?case* **using** *empty-fset-if-mdeg-le1-not-single* **by** *auto*
**qed**

**lemma** *normalize-full-forward-singleton*:
⟦*max-deg t1 ≤ 1*; *dom-children t1 T*; ∀ *v* ∈ *dverts t1*. *forward v*; *wf-dlverts t1*⟧
⟹ ∃ *r*. *normalize-full t1* = *Node r* {||} ∧ *forward r*
**using** *normalize-full-empty-sucs normalize-full-forward* **by** *fastforce*

**lemma** *denormalize-empty-sucs-simp*: *denormalize* (*Node r* {||}) = *r*
**using** *denormalize.simps*(*2*) **by** *blast*

**lemma** *normalize-full-dverts-eq-denormalize*:
**assumes** *max-deg t1 ≤ 1*
**shows** *dverts* (*normalize-full t1*) = {*denormalize t1*}
**proof** −
**obtain** *r* **where** *r-def*[*simp*]: *normalize-full t1* = *Node r* {||}
**using** *assms normalize-full-empty-sucs* **by** *blast*
**then have** *denormalize* (*normalize-full t1*) = *r* **by** (*simp add*: *denormalize-empty-sucs-simp*)
**then have** *r* = *denormalize t1* **using** *normalize-full-denormalize-eq* **by** *blast*
**then show** *?thesis* **by** *simp*
**qed**

**lemma** *normalize-full-normalize-dverts-eq-denormalize*:
**assumes** *wf-dlverts t1* **and** *max-deg t1 ≤ 1*
**shows** *dverts* (*normalize-full* (*normalize t1*)) = {*denormalize t1*}
**proof** −
**have** *max-deg* (*normalize t1*) ≤ *1* **using** *assms normalize-mdeg-eq′* **by** *fastforce*
**then show** *?thesis*
**using** *normalize-full-dverts-eq-denormalize normalize-denormalize-eq assms*(*1*)
**by** *simp*
**qed**

**lemma** *normalize-full-normalize-dverts-eq-denormalize′*:
**assumes** *wf-darcs t1* **and** *max-deg t1 ≤ 1*

**shows** *dverts* (*normalize-full* (*normalize t1*)) = {*denormalize t1*}
**proof** −
**have** *max-deg* (*normalize t1*) ≤ *1* **using** *assms normalize-mdeg-eq* **by** *fastforce*
**then show** *?thesis*
**using** *normalize-full-dverts-eq-denormalize normalize-denormalize-eq′ assms*(*1*)
**by** *simp*
**qed**

**lemma** *denormalize-full-forward*:
⟦*max-deg t1* ≤ *1*; *dom-children t1 T*; ∀ *v* ∈ *dverts t1*. *forward v*; *wf-dlverts t1*⟧
⟹ *forward* (*denormalize* (*normalize-full t1*))
**by** (*metis denormalize-empty-sucs-simp normalize-full-forward-singleton*)

**lemma** *denormalize-forward*:
⟦*max-deg t1* ≤ *1*; *dom-children t1 T*; ∀ *v* ∈ *dverts t1*. *forward v*; *wf-dlverts t1*⟧
⟹ *forward* (*denormalize t1*)
**using** *denormalize-full-forward* **by** *simp*

**lemma** *ikkbz-sub-forward-if-uneq*: *ikkbz-sub t* ≠ *t* ⟹ *forward* (*denormalize* (*ikkbz-sub t*))
**using** *denormalize-forward ikkbz-sub-mdeg-le1 dom-mdeg-le1-ikkbz-sub ikkbz-sub-wf-dlverts ranked-dtree-with-orig.verts-forward ikkbz-sub-ranked-dtree-orig*
**by** *fast*

**theorem** *ikkbz-sub-forward*:
⟦*max-deg t* ≤ *1* ⟹ *dom-children t T*⟧ ⟹ *forward* (*denormalize* (*ikkbz-sub t*))
**using** *ikkbz-sub-forward-if-uneq ikkbz-sub-eq-iff-mdeg-le1*[*of t*]
**by** (*fastforce simp*: *verts-forward wf-lverts denormalize-forward*)

**lemma** *root-arc-singleton*:
**assumes** *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T* **and** *wf-dlverts* (*Node r* {|(*t1*,*e1*)|})
**shows** ∃ *x*∈*set r*. ∃ *y*∈*set* (*Dtree.root t1*). *x* →$_T$ *y*
**using** *root-arc-if-dom-wfdlverts assms* **by** *fastforce*

**lemma** *before-if-dom-children-wf-conform*:
**assumes** *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T*
**and** ∀ *v* ∈ *dverts* (*Node r* {|(*t1*,*e1*)|}). *seq-conform v*
**and** *wf-dlverts* (*Node r* {|(*t1*,*e1*)|})
**shows** *before r* (*Dtree.root t1*)
**proof** −
**have** *seq-conform* (*Dtree.root t1*) **using** *dtree.set-sel*(*1*) *assms*(*2*) **by** *auto*
**moreover have** *seq-conform r* **using** *assms*(*2*) **by** *auto*
**moreover have** *set r* ∩ *set* (*Dtree.root t1*) = {}
**using** *assms*(*3*) *dlverts-eq-dverts-union dtree.set-sel*(*1*) **by** *fastforce*
**ultimately show** *?thesis* **unfolding** *before-def* **using** *root-arc-singleton assms*(*1*,*3*)
**by** *blast*
**qed**

**lemma** *root-arc-singleton′*:

**assumes** *Node r* $\{|(t1,e1)|\} = t$ **and** *dom-children t T*
**shows** $\exists x{\in}set\ r.\ \exists y{\in}set\ (Dtree.root\ t1).\ x \rightarrow_T y$
**using** *assms root-arc-singleton wf-lverts* **by** *blast*

**lemma** *root-before-if-dom*:
**assumes** *Node r* $\{|(t1,e1)|\} = t$ **and** *dom-children t T*
**shows** *before r* (*Dtree.root t1*)
**proof** −
**have** (*Dtree.root t1*) ∈ *dverts t* **using** *dtree.set-sel*(*1*) *assms*(*1*) **by** *fastforce*
**then have** *seq-conform* (*Dtree.root t1*) **using** *verts-conform* **by** *simp*
**moreover have** *seq-conform r* **using** *verts-conform assms*(*1*) **by** *auto*
**ultimately show** *?thesis*
**using** *before-def child-disjoint-root root-arc-singleton′ assms* **by** *fastforce*
**qed**

**lemma** *combine-conform*:
⟦*dom-children* (*Node r* $\{|(t1,e1)|\}$) *T*; ∀ *v* ∈ *dverts* (*Node r* $\{|(t1,e1)|\}$). *seq-conform v*;
*wf-dlverts* (*Node r* $\{|(t1,e1)|\}$); *v* ∈ *dverts* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))⟧
⟹ *seq-conform v*
**apply**(*cases v* = *r*@*Dtree.root t1*)
**using** *before-if-dom-children-wf-conform seq-conform-if-before* **apply** *fastforce*
**using** *dverts-suc-subseteq* **by** *fastforce*

**lemma** *denormalize-full-set-eq-dlverts*:
*max-deg t1* ≤ *1* ⟹ *set* (*denormalize* (*normalize-full t1*)) = *dlverts t1*
**using** *denormalize-set-eq-dlverts* **by** *auto*

**lemma** *denormalize-full-set-eq-dverts-union*:
*max-deg t1* ≤ *1* ⟹ *set* (*denormalize* (*normalize-full t1*)) = ⋃(*set* ‘ *dverts t1*)
**using** *denormalize-full-set-eq-dlverts dlverts-eq-dverts-union* **by** *fastforce*

**corollary** *hd-eq-denormalize-full*:
*wf-dlverts t1* ⟹ *hd* (*denormalize* (*normalize-full t1*)) = *hd* (*Dtree.root t1*)
**using** *denormalize-hd-root-wf* **by** *auto*

**corollary** *denormalize-full-nempty-if-wf*:
*wf-dlverts t1* ⟹ *denormalize* (*normalize-full t1*) ≠ []
**using** *denormalize-nempty-if-wf* **by** *auto*

**lemma** *take1-eq-denormalize-full*:
*wf-dlverts t1* ⟹ *take 1* (*denormalize* (*normalize-full t1*)) = [*hd* (*Dtree.root t1*)]
**using** *hd-eq-denormalize-full take1-eq-hd denormalize-full-nempty-if-wf* **by** *fast*

**lemma** *P-denormalize-full*:
**assumes** *wf-dlverts t1*
**and** ∀ *v* ∈ *dverts t1*. *distinct v*
**and** *hd* (*Dtree.root t1*) = *root*
**and** *max-deg t1* ≤ *1*

**shows** *unique-set-r root* (*dverts t1*) (*denormalize* (*normalize-full t1*))
**using** *assms unique-set-r-def denormalize-full-set-eq-dverts-union*
*denormalize-distinct normalize-full-wfdlverts take1-eq-denormalize-full*
**by** *fastforce*

**lemma** *P-denormalize*:
**fixes** *t1* :: (*'a list*,*'b*) *dtree*
**assumes** *wf-dlverts t1*
**and** ∀ *v* ∈ *dverts t1*. *distinct v*
**and** *hd* (*Dtree.root t1*) = *root*
**and** *max-deg t1* ≤ *1*
**shows** *unique-set-r root* (*dverts t1*) (*denormalize t1*)
**using** *assms P-denormalize-full* **by** *auto*

**lemma** *denormalize-full-fwd*:
**assumes** *wf-dlverts t1*
**and** *max-deg t1* ≤ *1*
**and** ∀ *xs* ∈ (*dverts t1*). *seq-conform xs*
**and** *dom-children t1 T*
**shows** *forward* (*denormalize* (*normalize-full t1*))
**using** *assms denormalize-forward forward-arcs-alt seq-conform-def* **by** *auto*

**lemma** *normalize-full-verts-sublist*:
*v* ∈ *dverts t1* ⟹ ∃ *v2* ∈ *dverts* (*normalize-full t1*). *sublist v v2*
**proof**(*induction t1 arbitrary*: *v rule*: *normalize-full.induct*)
**case** *ind*: (*1 r t e*)
**then consider** *v* = *r* ∨ *v* = *Dtree.root t* | ∃ *t1* ∈ *fst* ‘ *fset* (*sucs t*). *v* ∈ *dverts t1*
**using** *dverts-root-or-suc* **by** *fastforce*
**then show** *?case*
**proof**(*cases*)
**case** *1*
**have** ∃ *a*∈*dverts* (*normalize-full* (*Node* (*r* @ *Dtree.root t*) (*sucs t*))). *sublist* (*r*@*Dtree.root t*) *a*
**using** *ind.IH* **by** *simp*
**moreover have** *sublist v* (*r*@*Dtree.root t*) **using** *1* **by** *blast*
**ultimately show** *?thesis* **using** *sublist-order.dual-order.trans* **by** *auto*
**next**
**case** *2*
**then show** *?thesis* **using** *ind.IH*[*of v*] **by** *fastforce*
**qed**
**next**
**case** (*2 xs r*)
**then show** *?case* **by** *fastforce*
**qed**

**lemma** *normalize-full-sublist-preserv*:
⟦*sublist xs v*; *v* ∈ *dverts t1*⟧ ⟹ ∃ *v2* ∈ *dverts* (*normalize-full t1*). *sublist xs v2*
**using** *normalize-full-verts-sublist sublist-order.dual-order.trans* **by** *fast*

**lemma** *denormalize-full-sublist-preserv*:
**assumes** *sublist xs v* **and** *v* ∈ *dverts t1* **and** *max-deg t1* ≤ *1*
**shows** *sublist xs* (*denormalize* (*normalize-full t1*))
**proof** −
**obtain** *r* **where** *r-def*[*simp*]: *normalize-full t1* = *Node r* {||}
**using** *assms*(*3*) *normalize-full-empty-sucs* **by** *blast*
**have** *sublist xs r* **using** *normalize-full-sublist-preserv*[*OF assms*(*1*,*2*)] **by** *simp*
**then show** *?thesis* **by** (*simp add*: *denormalize-empty-sucs-simp*)
**qed**

**corollary** *denormalize-sublist-preserv*:
⟦*sublist xs v*; *v* ∈ *dverts* (*t1*::('*a list*,'*b*) *dtree*); *max-deg t1* ≤ *1*⟧
⟹ *sublist xs* (*denormalize t1*)
**using** *denormalize-full-sublist-preserv* **by** *simp*

**lemma** *Q-denormalize-full*:
**assumes** *wf-dlverts t1*
**and** ∀ *v* ∈ *dverts t1*. *distinct v*
**and** *hd* (*Dtree.root t1*) = *root*
**and** *max-deg t1* ≤ *1*
**and** ∀ *xs* ∈ (*dverts t1*). *seq-conform xs*
**and** *dom-children t1 T*
**shows** *fwd-sub root* (*dverts t1*) (*denormalize* (*normalize-full t1*))
**using** *P-denormalize-full*[*OF assms*(*1*−*4*)] *assms*(*1*,*4*−*6*) *denormalize-full-sublist-preserv*
**by** (*auto dest*: *denormalize-full-fwd simp*: *fwd-sub-def*)

**corollary** *Q-denormalize*:
**assumes** *wf-dlverts t1*
**and** ∀ *v* ∈ *dverts t1*. *distinct v*
**and** *hd* (*Dtree.root t1*) = *root*
**and** *max-deg t1* ≤ *1*
**and** ∀ *xs* ∈ (*dverts t1*). *seq-conform xs*
**and** *dom-children t1 T*
**shows** *fwd-sub root* (*dverts t1*) (*denormalize t1*)
**using** *Q-denormalize-full assms* **by** *simp*

**corollary** *Q-denormalize-t*:
**assumes** *hd* (*Dtree.root t*) = *root*
**and** *max-deg t* ≤ *1*
**and** *dom-children t T*
**shows** *fwd-sub root* (*dverts t*) (*denormalize t*)
**using** *Q-denormalize wf-lverts assms verts-conform verts-distinct* **by** *blast*

**lemma** *P-denormalize-ikkbz-sub*:
**assumes** *hd* (*Dtree.root t*) = *root*
**shows** *unique-set-r root* (*dverts t*) (*denormalize* (*ikkbz-sub t*))
**proof** −
**interpret** *T*: *ranked-dtree-with-orig ikkbz-sub t* **using** *ikkbz-sub-ranked-dtree-orig*
**by** *auto*

**have** $\forall v{\in}dverts\ (ikkbz\text{-}sub\ t).\ distinct\ v$ **using** *T.verts-distinct* **by** *simp*
**then show** *?thesis*
**using** *P-denormalize T.wf-lverts ikkbz-sub-mdeg-le1 assms ikkbz-sub-hd-root*
**unfolding** *unique-set-r-def denormalize-ikkbz-eq-dlverts dlverts-eq-dverts-union*
**by** *blast*
**qed**

**lemma** *merge1-sublist-preserv*:
$[\![sublist\ xs\ v;\ v \in dverts\ t]\!] \Longrightarrow \exists\, v2 \in dverts\ (merge1\ t).\ sublist\ xs\ v2$
**using** *sublist-order.dual-order.trans* **by** *auto*

**lemma** *normalize1-verts-sublist*: $v \in dverts\ t1 \Longrightarrow \exists\, v2 \in dverts\ (normalize1\ t1).$
$sublist\ v\ v2$
**proof**(*induction t1 arbitrary*: *v rule*: *normalize1.induct*)
**case** *ind*: (*1 r t e*)
**show** *?case*
**proof**(*cases rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*))
**case** *True*
**consider** $v = r \vee v = Dtree.root\ t \mid \exists\, t1 \in fst\ `\ fset\ (sucs\ t).\ v \in dverts\ t1$
**using** *dverts-root-or-suc* **using** *ind.prems* **by** *fastforce*
**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then show** *?thesis* **using** *True* **by** *auto*
**next**
**case** *2*
**then show** *?thesis* **using** *True* **by** *fastforce*
**qed**
**next**
**case** *False*
**then show** *?thesis* **using** *ind* **by** *auto*
**qed**
**next**
**case** (*2 xs r*)
**then show** *?case* **by** *fastforce*
**qed**

**lemma** *normalize1-sublist-preserv*:
$[\![sublist\ xs\ v;\ v \in dverts\ t1]\!] \Longrightarrow \exists\, v2 \in dverts\ (normalize1\ t1).\ sublist\ xs\ v2$
**using** *normalize1-verts-sublist sublist-order.dual-order.trans* **by** *fast*

**lemma** *normalize-verts-sublist*: $v \in dverts\ t1 \Longrightarrow \exists\, v2 \in dverts\ (normalize\ t1).$
$sublist\ v\ v2$
**proof**(*induction t1 arbitrary*: *v rule*: *normalize.induct*)
**case** (*1 t1*)
**then show** *?case*
**proof**(*cases t1 = normalize1 t1*)
**case** *True*
**then show** *?thesis* **using** *1.prems* **by** *auto*

**next**
**case** *False*
**then have** *eq*: *normalize* (*normalize1 t1*) = *normalize t1* **by** (*auto simp*: *Let-def*)
**then obtain** *v2* **where** *v2-def*: *v2* ∈ *dverts* (*normalize1 t1*) *sublist v v2*
**using** *normalize1-verts-sublist 1.prems* **by** *blast*
**then show** *?thesis*
**using** *1.IH*[*OF refl False v2-def*(*1*)] *eq sublist-order.dual-order.trans* **by** *auto*
**qed**
**qed**

**lemma** *normalize-sublist-preserv*:
⟦*sublist xs v*; *v* ∈ *dverts t1*⟧ ⟹ ∃ *v2* ∈ *dverts* (*normalize t1*). *sublist xs v2*
**using** *normalize-verts-sublist sublist-order.dual-order.trans* **by** *fast*

**lemma** *ikkbz-sub-verts-sublist*: *v* ∈ *dverts t* ⟹ ∃ *v2* ∈ *dverts* (*ikkbz-sub t*). *sublist v v2*
**using** *ranked-dtree-with-orig-axioms* **proof**(*induction t arbitrary*: *v rule*: *ikkbz-sub.induct*)
**case** (*1 t*)
**then interpret** *T*: *ranked-dtree-with-orig t* **by** *simp*
**interpret** *NT*: *ranked-dtree-with-orig normalize t*
**using** *T.ranked-dtree-orig-normalize* **by** *blast*
**show** *?case*
**proof**(*cases max-deg t* ≤ *1*)
**case** *True*
**then show** *?thesis* **using** *1.prems*(*1*) **by** *auto*
**next**
**case** *False*
**then have** *0*: ¬ (*max-deg t* ≤ *1* ∨ ¬ *list-dtree t*) **using** *T.list-dtree-axioms* **by** *auto*
**obtain** *v1* **where** *v1-def*: *v1* ∈ *dverts* (*normalize t*) *sublist v v1*
**using** *normalize-verts-sublist 1.prems*(*1*) **by** *blast*
**then have** *v1* ∈ *dverts* (*merge1* (*normalize t*)) **using** *NT.merge1-dverts-eq* **by** *blast*
**then obtain** *v2* **where** *v2-def*: *v2* ∈ *dverts* (*ikkbz-sub t*) *sublist v1 v2*
**using** *1 0 T.merge1-normalize-ranked-dtree-orig* **by** *force*
**then show** *?thesis* **using** *v1-def*(*2*) *sublist-order.dual-order.trans* **by** *blast*
**qed**
**qed**

**lemma** *ikkbz-sub-sublist-preserv*:
⟦*sublist xs v*; *v* ∈ *dverts t*⟧ ⟹ ∃ *v2* ∈ *dverts* (*ikkbz-sub t*). *sublist xs v2*
**using** *ikkbz-sub-verts-sublist sublist-order.dual-order.trans* **by** *fast*

**lemma** *denormalize-ikkbz-sub-verts-sublist*:
∀ *xs* ∈ (*dverts t*). *sublist xs* (*denormalize* (*ikkbz-sub t*))
**using** *ikkbz-sub-verts-sublist denormalize-sublist-preserv ikkbz-sub-mdeg-le1* **by** *blast*

**lemma** *denormalize-ikkbz-sub-sublist-preserv*:
⟦*sublist xs v*; *v* ∈ *dverts t*⟧ ⟹ *sublist xs* (*denormalize* (*ikkbz-sub t*))
**using** *denormalize-ikkbz-sub-verts-sublist sublist-order.dual-order.trans* **by** *blast*

**lemma** *Q-denormalize-ikkbz-sub*:
⟦*hd* (*Dtree.root t*) = *root*; *max-deg t* ≤ 1 ⟹ *dom-children t T*⟧
⟹ *fwd-sub root* (*dverts t*) (*denormalize* (*ikkbz-sub t*))
**using** *P-denormalize-ikkbz-sub ikkbz-sub-forward denormalize-ikkbz-sub-verts-sublist fwd-sub-def*
**by** *blast*

### 10.4.2 Minimal Cost of the result

**lemma** *normalize1-dverts-app-before-contr*:
⟦*v* ∈ *dverts* (*normalize1 t*); *v* ∉ *dverts t*⟧
⟹ ∃ *v1*∈*dverts t*. ∃ *v2*∈*dverts t*. *v1* @ *v2* = *v* ∧ *before v1 v2* ∧ *rank* (*rev v2*) < *rank* (*rev v1*)
**by** (*fastforce dest*: *normalize1-dverts-contr-subtree*
*simp*: *single-subtree-root-dverts single-subtree-child-root-dverts contr-before*)

**lemma** *normalize1-dverts-app-bfr-cntr-rnks*:
**assumes** *v* ∈ *dverts* (*normalize1 t*) **and** *v* ∉ *dverts t*
**shows** ∃ *U*∈*dverts t*. ∃ *V*∈*dverts t*. *U* @ *V* = *v* ∧ *before U V* ∧ *rank* (*rev V*) < *rank* (*rev U*)
∧ (∀ *xs* ∈ *dverts t*. (∃ *y*∈*set xs*. ¬ (∃ *x'*∈*set V*. *x'* →⁺ *T y*) ∧ (∃ *x*∈*set U*. *x* →⁺ *T y*) ∧ *xs* ≠ *U*)
⟶ *rank* (*rev V*) ≤ *rank* (*rev xs*))
**using** *normalize1-dverts-contr-subtree*[*OF assms*] *subtree-rank-ge-if-reach'*
**by** (*fastforce simp*: *single-subtree-root-dverts single-subtree-child-root-dverts contr-before*)

**lemma** *normalize1-dverts-app-bfr-cntr-rnks'*:
**assumes** *v* ∈ *dverts* (*normalize1 t*) **and** *v* ∉ *dverts t*
**shows** ∃ *U*∈*dverts t*. ∃ *V*∈*dverts t*. *U* @ *V* = *v* ∧ *before U V* ∧ *rank* (*rev V*) ≤ *rank* (*rev U*)
∧ (∀ *xs* ∈ *dverts t*. (∃ *y*∈*set xs*. ¬ (∃ *x'*∈*set V*. *x'* →⁺ *T y*) ∧ (∃ *x*∈*set U*. *x* →⁺ *T y*) ∧ *xs* ≠ *U*)
⟶ *rank* (*rev V*) ≤ *rank* (*rev xs*))
**using** *normalize1-dverts-contr-subtree*[*OF assms*] *subtree-rank-ge-if-reach'*
**by** (*fastforce simp*: *single-subtree-root-dverts single-subtree-child-root-dverts contr-before*)

**lemma** *normalize1-dverts-split*:
*dverts* (*normalize1 t1*)
= {*v* ∈ *dverts* (*normalize1 t1*). *v* ∉ *dverts t1*} ∪ {*v* ∈ *dverts* (*normalize1 t1*). *v* ∈ *dverts t1*}
**by** *blast*

**lemma** *normalize1-dlverts-split*:
*dlverts* (*normalize1 t1*)
= ⋃ (*set* ‘ {*v* ∈ *dverts* (*normalize1 t1*). *v* ∉ *dverts t1*})

$\cup \bigcup (set\ `\ \{v \in dverts\ (normalize1\ t1).\ v \in dverts\ t1\})$
**using** *dlverts-eq-dverts-union* **by** *fastforce*

**lemma** *normalize1-dsjnt-in-dverts*:
**assumes** *wf-dlverts t1*
**and** $v \in dverts\ t1$
**and** $set\ v \cap \bigcup (set\ `\ \{v \in dverts\ (normalize1\ t1).\ v \notin dverts\ t1\}) = \{\}$
**shows** $v \in dverts\ (normalize1\ t1)$
**proof** −
**have** $set\ v \subseteq dlverts\ (normalize1\ t1)$ **using** *assms*(2) *lverts-if-in-verts* **by** *fastforce*
**then have** *sub*: $set\ v \subseteq \bigcup (set\ `\ \{v \in dverts\ (normalize1\ t1).\ v \in dverts\ t1\})$
**using** *normalize1-dlverts-split assms*(3) **by** *auto*
**have** $v \neq []$ **using** *assms*(1,2) *empty-notin-wf-dlverts* **by** *auto*
**then obtain** *x* **where** *x-def*: $x \in set\ v$ **by** *fastforce*
**then show** *?thesis* **using** *dverts-same-if-set-wf*[*OF assms*(1,2)] *x-def sub* **by** *blast*
**qed**

**lemma** *normalize1-dsjnt-subset-split1*:
**fixes** *t1*
**defines** $X \equiv \{v \in dverts\ (normalize1\ t1).\ v \notin dverts\ t1\}$
**assumes** *wf-dlverts t1*
**shows** $\{x.\ x \in dverts\ t1 \wedge set\ x \cap \bigcup (set\ `\ X) = \{\}\} \subseteq \{v \in dverts\ (normalize1\ t1).\ v \in dverts\ t1\}$
**using** *assms normalize1-dsjnt-in-dverts* **by** *blast*

**lemma** *normalize1-dsjnt-subset-split2*:
**fixes** *t1*
**defines** $X \equiv \{v \in dverts\ (normalize1\ t1).\ v \notin dverts\ t1\}$
**assumes** *wf-dlverts t1*
**shows** $\{v \in dverts\ (normalize1\ t1).\ v \in dverts\ t1\} \subseteq \{x.\ x \in dverts\ t1 \wedge set\ x \cap \bigcup (set\ `\ X) = \{\}\}$
**using** *dverts-same-if-set-wf*[*OF wf-dlverts-normalize1*] *assms* **by** *blast*

**lemma** *normalize1-dsjnt-subset-eq-split*:
**fixes** *t1*
**defines** $X \equiv \{v \in dverts\ (normalize1\ t1).\ v \notin dverts\ t1\}$
**assumes** *wf-dlverts t1*
**shows** $\{v \in dverts\ (normalize1\ t1).\ v \in dverts\ t1\} = \{x.\ x \in dverts\ t1 \wedge set\ x \cap \bigcup (set\ `\ X) = \{\}\}$
**using** *normalize1-dsjnt-subset-split1 normalize1-dsjnt-subset-split2 assms*
**by** *blast*

**lemma** *normalize1-dverts-split2*:
**fixes** *t1*
**defines** $X \equiv \{v \in dverts\ (normalize1\ t1).\ v \notin dverts\ t1\}$
**assumes** *wf-dlverts t1*
**shows** $X \cup \{x.\ x \in dverts\ t1 \wedge set\ x \cap \bigcup (set\ `\ X) = \{\}\} = dverts\ (normalize1$

*t1*)
**unfolding** *assms*(*1*) **using** *normalize1-dsjnt-subset-eq-split*[*OF assms*(*2*)] **by** *blast*

**lemma** *set-subset-if-normalize1-vert*: *v1* ∈ *dverts* (*normalize1 t1*) ⟹ *set v1* ⊆ *dlverts t1*
**using** *lverts-if-in-verts* **by** *fastforce*

**lemma** *normalize1-new-verts-not-reach1*:
**assumes** *v1* ∈ *dverts* (*normalize1 t*) **and** *v1* ∉ *dverts t*
**and** *v2* ∈ *dverts* (*normalize1 t*) **and** *v2* ∉ *dverts t*
**and** *v1* ≠ *v2*
**shows** ¬(∃ *x*∈*set v1*. ∃ *y*∈*set v2*. *x* →⁺$_T$ *y*)
**using** *assms ranked-dtree-with-orig-axioms* **proof**(*induction t rule*: *normalize1.induct*)
**case** (*1 r t e*)
**then interpret** *R*: *ranked-dtree-with-orig Node r* {|(*t*, *e*)|} **by** *blast*
**show** *?case*
**proof**(*cases rank* (*rev* (*Dtree.root t*)) < *rank* (*rev r*))
**case** *True*
**then have** *eq*: *normalize1* (*Node r* {|(*t*, *e*)|}) = *Node* (*r*@*Dtree.root t*) (*sucs t*)
**by** *simp*
**have** *v1* = *r* @ *Dtree.root t*
**using** *1.prems*(*1*,*2*) *dverts-suc-subseteq* **unfolding** *eq* **by** *fastforce*
**moreover have** *v2* = *r* @ *Dtree.root t*
**using** *1.prems*(*3*,*4*) *dverts-suc-subseteq* **unfolding** *eq* **by** *fastforce*
**ultimately show** *?thesis* **using** *1.prems*(*5*) **by** *simp*
**next**
**case** *False*
**then show** *?thesis* **using** *1 R.ranked-dtree-orig-rec* **by** *simp*
**qed**
**next**
**case** (*2 xs r*)
**then interpret** *R*: *ranked-dtree-with-orig Node r xs* **by** *blast*
**have** *eq*: *normalize1* (*Node r xs*) = *Node r* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs*)
**using** *2.hyps* **by** *simp*
**obtain** *t1 e1* **where** *t1-def*: (*t1*,*e1*) ∈ *fset xs v1* ∈ *dverts* (*normalize1 t1*)
**using** *2.hyps 2.prems*(*1*,*2*) **by** *auto*
**obtain** *t2 e2* **where** *t2-def*: (*t2*,*e2*) ∈ *fset xs v2* ∈ *dverts* (*normalize1 t2*)
**using** *2.hyps 2.prems*(*3*,*4*) **by** *auto*
**show** *?case*
**proof**(*cases t1* = *t2*)
**case** *True*
**have** *v1* ∉ *dverts t1* ∧ *v2* ∉ *dverts t2*
**using** *2.hyps 2.prems*(*2*,*4*) *t1-def*(*1*) *t2-def*(*1*) **by** *simp*
**then show** *?thesis* **using** *2.IH t1-def t2-def True 2.prems*(*5*) *R.ranked-dtree-orig-rec*
**by** *simp*
**next**
**case** *False*
**have** *sub*: *is-subtree t1* (*Node r xs*) **using** *t1-def*(*1*) *subtree-if-child*[*of t1 xs r*]

**by** *force*
**have** *set v1* $\subseteq$ *dlverts t1* **using** *set-subset-if-normalize1-vert t1-def*(*2*) **by** *simp*
**then have** *reach-t1*: $\forall\, x \in$ *set v1*. $\forall\, y$. $x \rightarrow^+_T y \longrightarrow y \in$ *dlverts t1*
**using** *R.dlverts-reach1-in-dlverts sub* **by** *blast*
**have** *dlverts t1* $\cap$ *dlverts t2* $= \{\}$
**using** *R.wf-lverts t2-def*(*1*) *t1-def*(*1*) *wf-dlverts.simps*[*of r*] *False* **by** *fast*
**then have** *set v2* $\cap$ *dlverts t1* $= \{\}$ **using** *set-subset-if-normalize1-vert t2-def*(*2*)
**by** *auto*
**then show** *?thesis* **using** *reach-t1* **by** *blast*
**qed**
**qed**

**lemma** *normalize1-dverts-split-optimal*:
**defines** $X \equiv \{v \in$ *dverts* (*normalize1 t*). $v \notin$ *dverts t*$\}$
**assumes** $\exists\, x$. *fwd-sub root* (*dverts t*) *x*
**shows** $\exists\, zs$. *fwd-sub root* ($X \cup \{x.\ x \in$ *dverts t* $\wedge$ *set x* $\cap \bigcup$(*set* ‘ *X*) $= \{\}\}$) *zs*
$\wedge$ ($\forall$ *as*. *fwd-sub root* (*dverts t*) *as* $\longrightarrow$ *cost* (*rev zs*) $\leq$ *cost* (*rev as*))
**proof** −
**let** *?Y* = *dverts t*
**have** *dsjt*: $\forall$ *xs* $\in$ *?Y*. $\forall$ *ys* $\in$ *?Y*. *xs* = *ys* $\vee$ *set xs* $\cap$ *set ys* $= \{\}$
**using** *dverts-same-if-set-wf*[*OF wf-lverts*] **by** *blast*
**have** *fwd*: $\forall$ *xs* $\in$ *?Y*. *forward xs* **by** (*simp add*: *verts-forward*)
**have** *nempty*: [] $\notin$ *?Y* **by** (*simp add*: *empty-notin-wf-dlverts wf-lverts*)
**have** *fin*: *finite ?Y* **by** (*simp add*: *finite-dverts*)
**have** $\forall$ *ys* $\in$ *X*. $\exists$ *U* $\in$ *?Y*. $\exists$ *V* $\in$ *?Y*. *U*@*V* = *ys* $\wedge$ *before U V* $\wedge$ *rank* (*rev V*) $\leq$ *rank* (*rev U*)
$\wedge$ ($\forall$ *xs* $\in$ *?Y*. ($\exists$ *y*$\in$*set xs*. $\neg$($\exists$ *x*′$\in$*set V*. *x*′ $\rightarrow^+_T$ *y*) $\wedge$ ($\exists$ *x*$\in$*set U*. *x* $\rightarrow^+_T$ *y*) $\wedge$ *xs* $\neq$ *U*)
$\longrightarrow$ *rank* (*rev V*) $\leq$ *rank* (*rev xs*))
**unfolding** *X-def* **using** *normalize1-dverts-app-bfr-cntr-rnks*′ **by** *blast*
**moreover have** $\forall$ *xs* $\in$ *X*. $\forall$ *ys* $\in$ *X*. *xs* = *ys* $\vee$ *set xs* $\cap$ *set ys* $= \{\}$
**unfolding** *X-def* **using** *dverts-same-if-set-wf*[*OF wf-dlverts-normalize1*] *wf-lverts*
**by** *blast*
**moreover have** $\forall$ *xs* $\in$ *X*. $\forall$ *ys* $\in$ *X*. *xs* = *ys* $\vee$ $\neg$($\exists$ *x*$\in$*set xs*. $\exists$ *y*$\in$*set ys*. *x* $\rightarrow^+_T$ *y*)
**unfolding** *X-def* **using** *normalize1-new-verts-not-reach1* **by** *blast*
**moreover have** *finite X* **by** (*simp add*: *X-def finite-dverts*)
**ultimately show** *?thesis*
**using** *combine-union-sets-optimal-cost*[*OF asi-rank dsjt fwd nempty fin assms*(*2*)]
**by** *simp*
**qed**

**corollary** *normalize1-dverts-optimal*:
**assumes** $\exists\, x$. *fwd-sub root* (*dverts t*) *x*
**shows** $\exists\, zs$. *fwd-sub root* (*dverts* (*normalize1 t*)) *zs*
$\wedge$ ($\forall$ *as*. *fwd-sub root* (*dverts t*) *as* $\longrightarrow$ *cost* (*rev zs*) $\leq$ *cost* (*rev as*))
**using** *normalize1-dverts-split-optimal assms normalize1-dverts-split2*[*OF wf-lverts*]
**by** *simp*

**lemma** *normalize-dverts-optimal*:
**assumes** $\exists x.$ *fwd-sub root* (*dverts t*) *x*
**shows** $\exists zs.$ *fwd-sub root* (*dverts* (*normalize t*)) *zs*
$\wedge$ ($\forall as.$ *fwd-sub root* (*dverts t*) *as* $\longrightarrow$ *cost* (*rev zs*) $\leq$ *cost* (*rev as*))
**using** *assms ranked-dtree-with-orig-axioms* **proof**(*induction t rule*: *normalize.induct*)
**case** (*1 t*)
**then interpret** *T*: *ranked-dtree-with-orig t* **by** *blast*
**obtain** *zs* **where** *zs-def*:
*fwd-sub root* (*dverts* (*normalize1 t*)) *zs*
$\forall as.$ *fwd-sub root* (*dverts t*) *as* $\longrightarrow$ *cost* (*rev zs*) $\leq$ *cost* (*rev as*)
**using** *1.prems T.normalize1-dverts-optimal* **by** *auto*
**show** *?case*
**proof**(*cases t* = *normalize1 t*)
**case** *True*
**then show** *?thesis* **using** *zs-def* **by** *auto*
**next**
**case** *False*
**then have** *eq*: *normalize* (*normalize1 t*) = *normalize t* **by** (*auto simp*: *Let-def*)
**have** $\exists zs.$ *fwd-sub root* (*dverts* (*normalize* (*normalize1 t*))) *zs*
$\wedge$ ($\forall as.$ *fwd-sub root* (*dverts* (*normalize1 t*)) *as* $\longrightarrow$ *cost* (*rev zs*) $\leq$ *cost* (*rev as*))
**using** *1.IH False zs-def*(*1*) *T.ranked-dtree-orig-normalize1* **by** *blast*
**then show** *?thesis* **using** *zs-def eq* **by** *force*
**qed**
**qed**

**lemma** *merge1-dverts-optimal*:
**assumes** $\exists x.$ *fwd-sub root* (*dverts t*) *x*
**shows** $\exists zs.$ *fwd-sub root* (*dverts* (*merge1 t*)) *zs*
$\wedge$ ($\forall as.$ *fwd-sub root* (*dverts t*) *as* $\longrightarrow$ *cost* (*rev zs*) $\leq$ *cost* (*rev as*))
**using** *assms forward-UV-lists-argmin-ex* **by** *simp*

**theorem** *ikkbz-sub-dverts-optimal*:
**assumes** $\exists x.$ *fwd-sub root* (*dverts t*) *x*
**shows** $\exists zs.$ *fwd-sub root* (*dverts* (*ikkbz-sub t*)) *zs*
$\wedge$ ($\forall as.$ *fwd-sub root* (*dverts t*) *as* $\longrightarrow$ *cost* (*rev zs*) $\leq$ *cost* (*rev as*))
**using** *assms ranked-dtree-with-orig-axioms* **proof**(*induction t rule*: *ikkbz-sub.induct*)
**case** (*1 t*)
**then interpret** *T*: *ranked-dtree-with-orig t* **by** *simp*
**interpret** *NT*: *ranked-dtree-with-orig normalize t*
**using** *T.ranked-dtree-orig-normalize* **by** *blast*
**show** *?case*
**proof**(*cases max-deg t* $\leq$ *1*)
**case** *True*
**then show** *?thesis* **using** *1.prems*(*1*) *forward-UV-lists-argmin-ex* **by** *auto*
**next**
**case** *False*
**then have** *0*: $\neg$ (*max-deg t* $\leq$ *1* $\vee$ $\neg$ *list-dtree t*) **using** *T.list-dtree-axioms* **by** *auto*

**obtain** *zs* **where** *zs-def*: *fwd-sub root* (*dverts* (*merge1* (*normalize t*))) *zs*
∀ *as*. *fwd-sub root* (*dverts t*) *as* ⟶ *cost* (*rev zs*) ≤ *cost* (*rev as*)
**using** *1.prems T.normalize-dverts-optimal NT.merge1-dverts-eq* **by** *auto*
**have** ∃ *zs*. *fwd-sub root* (*dverts* (*ikkbz-sub* (*merge1* (*normalize t*)))) *zs*
∧ (∀ *as*. *fwd-sub root* (*dverts* (*merge1* (*normalize t*))) *as* ⟶ *cost* (*rev zs*) ≤ *cost* (*rev as*))
**using** *1.IH 0 zs-def*(*1*) *T.merge1-normalize-ranked-dtree-orig* **by** *blast*
**then show** *?thesis* **using** *zs-def 0* **by** *force*
**qed**
**qed**

**lemma** *ikkbz-sub-dverts-optimal'*:
**assumes** *hd* (*Dtree.root t*) = *root* **and** *max-deg t* ≤ *1* ⟹ *dom-children t T*
**shows** ∃ *zs*. *fwd-sub root* (*dverts* (*ikkbz-sub t*)) *zs*
∧ (∀ *as*. *fwd-sub root* (*dverts t*) *as* ⟶ *cost* (*rev zs*) ≤ *cost* (*rev as*))
**using** *ikkbz-sub-dverts-optimal Q-denormalize-ikkbz-sub assms* **by** *blast*

**lemma** *combine-strict-subtree-orig*:
**assumes** *strict-subtree* (*Node r1* {|(*t2*,*e2*)|}) (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
**shows** *is-subtree* (*Node r1* {|(*t2*,*e2*)|}) (*Node r* {|(*t1*,*e1*)|})
**proof** −
**obtain** *t3* **where** *t3-def*: *t3* ∈ *fst* ‘ *fset* (*sucs t1*) *is-subtree* (*Node r1* {|(*t2*,*e2*)|}) *t3*
**using** *assms* **unfolding** *strict-subtree-def* **by** *force*
**then show** *?thesis* **using** *subtree-trans subtree-if-suc*[*OF t3-def*(*1*)] **by** *auto*
**qed**

**lemma** *combine-subtree-orig-uneq*:
**assumes** *is-subtree* (*Node r1* {|(*t2*,*e2*)|}) (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
**shows** *Node r1* {|(*t2*,*e2*)|} ≠ *Node r* {|(*t1*,*e1*)|}
**proof** −
**have** *size* (*Node r1* {|(*t2*,*e2*)|}) ≤ *size* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
**using** *assms*(*1*) *subtree-size-le* **by** *blast*
**also have** *size* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*)) < *size* (*Node r* {|(*t1*,*e1*)|})
**using** *dtree-size-skip-decr1* **by** *fast*
**finally show** *?thesis* **by** *blast*
**qed**

**lemma** *combine-strict-subtree-ranks-le*:
**assumes** ⋀*r1 t2 e2*. *strict-subtree* (*Node r1* {|(*t2*,*e2*)|}) (*Node r* {|(*t1*,*e1*)|})
⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*Dtree.root t2*))
**and** *strict-subtree* (*Node r1* {|(*t2*,*e2*)|}) (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
**shows** *rank* (*rev r1*) ≤ *rank* (*rev* (*Dtree.root t2*))
**using** *combine-strict-subtree-orig assms* **unfolding** *strict-subtree-def*
**by** (*fast intro*!: *combine-subtree-orig-uneq* )

**lemma** *subtree-child-uneq*:
⟦*is-subtree t1 t2*; *t2* ∈ *fst* ‘ *fset xs*⟧ ⟹ *t1* ≠ *Node r xs*
**using** *child-uneq subtree-antisym subtree-if-child* **by** *fast*

**lemma** *subtree-singleton-child-uneq*:
*is-subtree t1 t2* $\Longrightarrow$ *t1* $\neq$ *Node r* $\{|(t2,e2)|\}$
**using** *subtree-child-uneq*[*of t1*] **by** *simp*

**lemma** *child-subtree-ranks-le-if-strict-subtree*:
**assumes** $\bigwedge$*r1 t2 e2*. *strict-subtree* (*Node r1* $\{|(t2,e2)|\}$) (*Node r* $\{|(t1,e1)|\}$)
$\Longrightarrow$ *rank* (*rev r1*) $\leq$ *rank* (*rev* (*Dtree.root t2*))
**and** *is-subtree* (*Node r1* $\{|(t2,e2)|\}$) *t1*
**shows** *rank* (*rev r1*) $\leq$ *rank* (*rev* (*Dtree.root t2*))
**using** *assms subtree-trans subtree-singleton-child-uneq* **unfolding** *strict-subtree-def*
**by** *fastforce*

**lemma** *verts-ge-child-if-sorted*:
**assumes** $\bigwedge$*r1 t2 e2*. *strict-subtree* (*Node r1* $\{|(t2,e2)|\}$) (*Node r* $\{|(t1,e1)|\}$)
$\Longrightarrow$ *rank* (*rev r1*) $\leq$ *rank* (*rev* (*Dtree.root t2*))
**and** *max-deg* (*Node r* $\{|(t1,e1)|\}$) $\leq$ *1*
**and** *v* $\in$ *dverts t1*
**shows** *rank* (*rev* (*Dtree.root t1*)) $\leq$ *rank* (*rev v*)
**proof** $-$
**have** $\bigwedge$*r1 t2 e2*. *is-subtree* (*Node r1* $\{|(t2,e2)|\}$) *t1* $\Longrightarrow$ *rank* (*rev r1*) $\leq$ *rank* (*rev* (*Dtree.root t2*))
**using** *child-subtree-ranks-le-if-strict-subtree*[*OF assms*(*1*)] **by** *simp*
**moreover have** *max-deg t1* $\leq$ *1* **using** *mdeg-ge-child*[*of t1 e1* $\{|(t1,e1)|\}$] *assms*(*2*) **by** *simp*
**ultimately show** *?thesis* **using** *rank-ge-if-mdeg-le1-dvert-nocontr assms*(*3*) **by** *fastforce*
**qed**

**lemma** *verts-ge-child-if-sorted'*:
**assumes** $\bigwedge$*r1 t2 e2*. *strict-subtree* (*Node r1* $\{|(t2,e2)|\}$) (*Node r* $\{|(t1,e1)|\}$)
$\Longrightarrow$ *rank* (*rev r1*) $\leq$ *rank* (*rev* (*Dtree.root t2*))
**and** *max-deg* (*Node r* $\{|(t1,e1)|\}$) $\leq$ *1*
**and** *v* $\in$ *dverts* (*Node r* $\{|(t1,e1)|\}$)
**and** *v* $\neq$ *r*
**shows** *rank* (*rev* (*Dtree.root t1*)) $\leq$ *rank* (*rev v*)
**using** *verts-ge-child-if-sorted*[*OF assms*(*1,2*)] *assms*(*3,4*) **by** *simp*

**lemma** *not-combined-sub-dverts-combine*:
$\{r$@*Dtree.root t1*$\}$ $\cup$ $\{x.\ x \in$ *dverts* (*Node r* $\{|(t1,e1)|\}$) $\wedge$ *x* $\neq$ *r* $\wedge$ *x* $\neq$ *Dtree.root t1*$\}$
$\subseteq$ *dverts* (*Node* (*r* @ *Dtree.root t1*) (*sucs t1*))
**using** *dverts-suc-subseteq dverts-root-or-suc* **by** *fastforce*

**lemma** *dverts-combine-orig-not-combined*:
**assumes** *wf-dlverts* (*Node r* $\{|(t1,e1)|\}$) **and** *x* $\in$ *dverts* (*Node* (*r* @ *Dtree.root t1*) (*sucs t1*)) **and** *x* $\neq$ *r*@*Dtree.root t1*
**shows** *x* $\in$ *dverts* (*Node r* $\{|(t1,e1)|\}$) $\wedge$ *x* $\neq$ *r* $\wedge$ *x* $\neq$ *Dtree.root t1*
**proof** $-$

**obtain** *t2* **where** *t2-def*: *t2* ∈ *fst* ‘ *fset* (*sucs t1*) *x* ∈ *dverts t2* **using** *assms*(*2*,*3*)
**by** *fastforce*
**have** *set r* ∩ *dlverts t2* = {} **using** *assms*(*1*) *suc-in-dlverts'*[*OF t2-def*(*1*)] **by**
*auto*
**then have** *x* ≠ *r* **using** *assms*(*1*) *t2-def*(*2*) *nempty-inter-notin-dverts* **by** *auto*
**have** *Dtree.root t1* ≠ []
**using** *assms*(*1*) *empty-notin-wf-dlverts single-subtree-child-root-dverts*[*OF self-subtree*,
*of t1*]
**by** *force*
**moreover have** *set* (*Dtree.root t1*) ∩ *dlverts t2* = {}
**using** *assms*(*1*) *t2-def*(*1*) *notin-dlverts-suc-if-wf-in-root* **by** *fastforce*
**ultimately have** *x* ≠ *Dtree.root t1* **using** *nempty-inter-notin-dverts t2-def*(*2*)
**by** *blast*
**then show** *?thesis* **using** ‹*x* ≠ *r*› *t2-def dverts-suc-subseteq* **by** *auto*
**qed**

**lemma** *dverts-combine-sub-not-combined*:
*wf-dlverts* (*Node r* {|(*t1*,*e1*)|}) ⟹ *dverts* (*Node* (*r* @ *Dtree.root t1*) (*sucs t1*))
⊆ {*r*@*Dtree.root t1*} ∪ {*x*. *x* ∈ *dverts* (*Node r* {|(*t1*,*e1*)|}) ∧ *x* ≠ *r* ∧ *x* ≠
*Dtree.root t1*}
**using** *dverts-combine-orig-not-combined* **by** *fast*

**lemma** *dverts-combine-eq-not-combined*:
*wf-dlverts* (*Node r* {|(*t1*,*e1*)|}) ⟹ *dverts* (*Node* (*r* @ *Dtree.root t1*) (*sucs t1*))
= {*r*@*Dtree.root t1*} ∪ {*x*. *x* ∈ *dverts* (*Node r* {|(*t1*,*e1*)|}) ∧ *x* ≠ *r* ∧ *x* ≠
*Dtree.root t1*}
**using** *dverts-combine-sub-not-combined not-combined-sub-dverts-combine* **by** *fast*

**lemma** *normalize-full-dverts-optimal-if-sorted*:
**assumes** *asi rank root cost*
**and** *wf-dlverts t1*
**and** ∀ *xs* ∈ (*dverts t1*). *distinct xs*
**and** ∀ *xs* ∈ (*dverts t1*). *seq-conform xs*
**and** ⋀*r1 t2 e2*. *strict-subtree* (*Node r1* {|(*t2*,*e2*)|}) *t1*
⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*Dtree.root t2*))
**and** *max-deg t1* ≤ *1*
**and** *hd* (*Dtree.root t1*) = *root*
**and** *dom-children t1 T*
**shows** ∃ *zs*. *fwd-sub root* (*dverts* (*normalize-full t1*)) *zs*
∧ (∀ *as*. *fwd-sub root* (*dverts t1*) *as* ⟶ *cost* (*rev zs*) ≤ *cost* (*rev as*))
**using** *assms* **proof**(*induction t1 rule*: *normalize-full.induct*)
**case** (*1 r t e*)
**let** *?Y* = *dverts* (*Node r* {|(*t*,*e*)|})
**have** *dsjt*: ∀ *xs* ∈ *?Y*. ∀ *ys* ∈ *?Y*. *xs* = *ys* ∨ *set xs* ∩ *set ys* = {}
**using** *dverts-same-if-set-wf*[*OF 1.prems*(*2*)] **by** *blast*
**have** *fwd*: ∀ *xs* ∈ *?Y*. *forward xs* **using** *1.prems*(*4*) *seq-conform-alt* **by** *blast*
**have** *nempty*: [] ∉ *?Y* **using** *empty-notin-wf-dlverts 1.prems*(*2*) **by** *blast*
**have** *fin*: *finite ?Y* **by** (*simp add*: *finite-dverts*)
**have** *U*: *r* ∈ *dverts* (*Node r* {|(*t*, *e*)|}) **by** *simp*

**have** *V*: *Dtree.root t* ∈ *dverts* (*Node r* {|(*t*, *e*)|})
**using** *single-subtree-child-root-dverts self-subtree* **by** *fast*
**have** *ge*: ∀ *xs*∈*dverts* (*Node r* {|(*t*, *e*)|}). *xs* ≠ *r* ⟶ *rank* (*rev* (*Dtree.root t*)) ≤ *rank* (*rev xs*)
**using** *verts-ge-child-if-sorted'*[*OF 1.prems*(*5*,*6*)] **by** *fast*
**moreover have** *bfr*: *before r* (*Dtree.root t*)
**using** *before-if-dom-children-wf-conform*[*OF 1.prems*(*8*,*4*,*2*)].
**moreover have** *Ex*: ∃ *x*. *fwd-sub root ?Y x* **using** *Q-denormalize-full 1.prems*(*1*−*8*) **by** *blast*
**ultimately obtain** *zs* **where** *zs-def*:
*fwd-sub root* ({*r*@*Dtree.root t*} ∪ {*x*. *x* ∈ *?Y* ∧ *x* ≠ *r* ∧ *x* ≠ *Dtree.root t*}) *zs*
(∀ *as*. *fwd-sub root ?Y as* ⟶ *cost* (*rev zs*) ≤ *cost* (*rev as*))
**using** *app-UV-set-optimal-cost*[*OF 1.prems*(*1*) *dsjt fwd nempty fin U V*] **by** *blast*
**have** *wf*: *wf-dlverts* (*Node* (*r* @ *Dtree.root t*) (*sucs t*)) **using** *1.prems*(*2*) *combine-wf-dlverts* **by** *fast*
**moreover have** *dst*: ∀ *v*∈*dverts* (*Node* (*r* @ *Dtree.root t*) (*sucs t*)). *distinct v*
**using** *1.prems*(*2*,*3*) *combine-distinct* **by** *fast*
**moreover have** *seq*: ∀ *v*∈*dverts* (*Node* (*r* @ *Dtree.root t*) (*sucs t*)). *seq-conform v*
**using** *1.prems*(*2*,*4*,*8*) *combine-conform* **by** *blast*
**moreover have** *rnk*: ⋀*r1 t2 e2*. *strict-subtree* (*Node r1* {|(*t2*,*e2*)|}) (*Node* (*r* @ *Dtree.root t*) (*sucs t*))
⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*Dtree.root t2*))
**using** *combine-strict-subtree-ranks-le*[*OF 1.prems*(*5*)] **by** *simp*
**moreover have** *mdeg*: *max-deg* (*Node* (*r* @ *Dtree.root t*) (*sucs t*)) ≤ *1*
**using** *1.prems*(*6*) *mdeg-child-sucs-le*
**by** (*fastforce dest*: *order-trans simp del*: *max-deg.simps*)
**moreover have** *hd*: *hd* (*Dtree.root* (*Node* (*r* @ *Dtree.root t*) (*sucs t*))) = *root*
**using** *1.prems*(*2*,*7*) **by** *simp*
**moreover have** *dom*: *dom-children* (*Node* (*r* @ *Dtree.root t*) (*sucs t*)) *T*
**using** *1.prems*(*8*) *dom-children-combine* **by** *auto*
**ultimately obtain** *xs* **where** *xs-def*:
*fwd-sub root* (*dverts* (*normalize-full* (*Node* (*r* @ *Dtree.root t*) (*sucs t*)))) *xs*
(∀ *as*. *fwd-sub root* (*dverts* (*Node* (*r* @ *Dtree.root t*) (*sucs t*))) *as*
⟶ *cost* (*rev xs*) ≤ *cost* (*rev as*))
**using** *1.IH 1.prems*(*1*) **by** *blast*
**then show** *?case* **using** *dverts-combine-eq-not-combined*[*OF 1.prems*(*2*)] *zs-def* **by** *force*
**next**
**case** (*2 xs r*)
**have** *Ex*: ∃ *x*. *fwd-sub root* (*dverts* (*Node r xs*)) *x*
**using** *Q-denormalize-full 2.prems*(*1*−*8*) **by** *blast*
**then show** *?case* **using** *2.hyps*(*1*) *forward-UV-lists-argmin-ex* **by** *simp*
**qed**

**corollary** *normalize-full-dverts-optimal-if-sorted'*:
**assumes** *max-deg t* ≤ *1*
**and** *hd* (*Dtree.root t*) = *root*

**and** *dom-children t T*
**and** $\bigwedge$*r1 t2 e2. strict-subtree* (*Node r1* {|(*t2*,*e2*)|}) *t*
$\Longrightarrow$ *rank* (*rev r1*) $\leq$ *rank* (*rev* (*Dtree.root t2*))
**shows** $\exists$ *zs. fwd-sub root* (*dverts* (*normalize-full t*)) *zs*
$\wedge$ ($\forall$ *as. fwd-sub root* (*dverts t*) *as* $\longrightarrow$ *cost* (*rev zs*) $\leq$ *cost* (*rev as*))
**using** *normalize-full-dverts-optimal-if-sorted asi-rank wf-lverts assms*
**by** (*blast intro*: *verts-distinct verts-conform*)

**lemma** *normalize-full-normalize-dverts-optimal*:
**assumes** *max-deg t* $\leq$ *1*
**and** *hd* (*Dtree.root t*) = *root*
**and** *dom-children t T*
**shows** $\exists$ *zs. fwd-sub root* (*dverts* (*normalize-full* (*normalize t*))) *zs*
$\wedge$ ($\forall$ *as. fwd-sub root* (*dverts t*) *as* $\longrightarrow$ *cost* (*rev zs*) $\leq$ *cost* (*rev as*))
**proof** −
**interpret** *NT*: *ranked-dtree-with-orig normalize t*
**using** *ranked-dtree-orig-normalize* **by** *auto*
**have** *mdeg*: *max-deg* (*normalize t*) $\leq$ *1* **using** *assms*(*1*) *normalize-mdeg-eq wf-arcs*
**by** *fastforce*
**moreover from** *this* **have** *dom*: *dom-children* (*normalize t*) *T*
**using** *assms*(*3*) *dom-mdeg-le1-normalize* **by** *fastforce*
**moreover have** *hd*: *hd* (*Dtree.root* (*normalize t*)) = *root*
**using** *assms*(*2*) *normalize-hd-root-eq′ wf-lverts* **by** *blast*
**moreover have** $\bigwedge$*r1 t2 e2*. ⟦*is-subtree* (*Node r1* {|(*t2*,*e2*)|}) (*normalize t*)⟧
$\Longrightarrow$ *rank* (*rev r1*) $\leq$ *rank* (*rev* (*Dtree.root t2*))
**by** (*simp add*: *normalize-sorted-ranks*)
**ultimately obtain** *xs* **where** *xs-def*: *fwd-sub root* (*dverts* (*normalize-full* (*normalize t*))) *xs*
($\forall$ *as. fwd-sub root* (*dverts* (*normalize t*)) *as* $\longrightarrow$ *cost* (*rev xs*) $\leq$ *cost* (*rev as*))
**using** *NT.normalize-full-dverts-optimal-if-sorted′ strict-subtree-def* **by** *blast*
**obtain** *zs* **where** *zs-def*: *fwd-sub root* (*dverts* (*normalize t*)) *zs*
($\forall$ *as. fwd-sub root* (*dverts t*) *as* $\longrightarrow$ *cost* (*rev zs*) $\leq$ *cost* (*rev as*))
**using** *normalize-dverts-optimal Q-denormalize-t assms* **by** *blast*
**then show** *?thesis* **using** *xs-def* **by** *force*
**qed**

**lemma** *single-set-distinct-sublist*: ⟦*set ys* = *set x*; *distinct ys*; *sublist x ys*⟧ $\Longrightarrow$ *x* = *ys*
**unfolding** *sublist-def*
**by** (*metis DiffD2 append.assoc append.left-neutral append.right-neutral list.set-intros*(*1*) *append-Cons distinct-set-diff neq-Nil-conv distinct-app-trans-l*)

**lemma** *denormalize-optimal-if-mdeg-le1*:
**assumes** *max-deg t* $\leq$ *1* **and** *hd* (*Dtree.root t*) = *root* **and** *dom-children t T*
**shows** $\forall$ *as. fwd-sub root* (*dverts t*) *as* $\longrightarrow$ *cost* (*rev* (*denormalize t*)) $\leq$ *cost* (*rev as*)
**proof** −
**obtain** *zs* **where** *zs-def*: *fwd-sub root* (*dverts* (*normalize-full* (*normalize t*))) *zs*
($\forall$ *as. fwd-sub root* (*dverts t*) *as* $\longrightarrow$ *cost* (*rev zs*) $\leq$ *cost* (*rev as*))

**using** *normalize-full-normalize-dverts-optimal assms* **by** *blast*
**have** *dverts* (*normalize-full* (*normalize t*)) = {*denormalize t*}
**using** *normalize-full-normalize-dverts-eq-denormalize wf-lverts assms*(*1*) **by** *blast*
**then show** *?thesis*
**using** *zs-def single-set-distinct-sublist* **by** (*auto simp*: *fwd-sub-def unique-set-r-def*)
**qed**

**theorem** *denormalize-ikkbz-sub-optimal*:
**assumes** *hd* (*Dtree.root t*) = *root* **and** *max-deg t* ≤ *1* ⟹ *dom-children t T*
**shows** (∀ *as*. *fwd-sub root* (*dverts t*) *as*
⟶ *cost* (*rev* (*denormalize* (*ikkbz-sub t*))) ≤ *cost* (*rev as*))
**proof** −
**obtain** *zs* **where** *zs-def*: *fwd-sub root* (*dverts* (*ikkbz-sub t*)) *zs*
∀ *as*. *fwd-sub root* (*dverts t*) *as* ⟶ *cost* (*rev zs*) ≤ *cost* (*rev as*)
**using** *ikkbz-sub-dverts-optimal′ assms* **by** *blast*
**interpret** *T*: *ranked-dtree-with-orig ikkbz-sub t* **using** *ikkbz-sub-ranked-dtree-orig* **by** *simp*
**have** *max-deg* (*ikkbz-sub t*) ≤ *1* **using** *ikkbz-sub-mdeg-le1* **by** *auto*
**have** *hd* (*Dtree.root* (*ikkbz-sub t*)) = *root* **using** *assms*(*1*) *ikkbz-sub-hd-root* **by** *auto*
**moreover have** *dom-children* (*ikkbz-sub t*) *T*
**using** *assms*(*2*) *dom-mdeg-le1-ikkbz-sub ikkbz-sub-eq-iff-mdeg-le1* **by** *auto*
**ultimately have** ∀ *as*. *fwd-sub root* (*dverts* (*ikkbz-sub t*)) *as*
⟶ *cost* (*rev* (*denormalize* (*ikkbz-sub t*))) ≤ *cost* (*rev as*)
**using** *T.denormalize-optimal-if-mdeg-le1*[*OF ikkbz-sub-mdeg-le1*] **by** *blast*
**then show** *?thesis* **using** *zs-def order-trans* **by** *blast*
**qed**

**end**

## 10.5 Arc Invariants hold for Conversion to Dtree

**context** *precedence-graph*
**begin**

**interpretation** *t*: *ranked-dtree to-list-dtree* **by** (*rule to-list-dtree-ranked-dtree*)

**lemma** *subtree-to-list-dtree-tree-dom*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; *t* ∈ *fst* ‘ *fset xs*⟧ ⟹ *r* →$_{to\text{-}list\text{-}tree}$ *Dtree.root t*
**unfolding** *to-list-dtree-def*
**using** *finite-directed-tree.subtree-child-dom to-list-tree-finite-directed-tree* **by** *fastforce*

**lemma** *subtree-to-list-dtree-dom*:
**assumes** *is-subtree* (*Node r xs*) *to-list-dtree* **and** *t* ∈ *fst* ‘ *fset xs*
**shows** *hd r* →$_T$ *hd* (*Dtree.root t*)
**proof** −

**interpret** $T$: *directed-tree to-list-tree* [*root*] **by** (*rule to-list-tree-directed-tree*)
**have** *0*: $r \rightarrow_{to\text{-}list\text{-}tree} Dtree.root\ t$ **using** *subtree-to-list-dtree-tree-dom assms* **by** *blast*
**then obtain** $x$ **where** *x-def*: $r = [x] \wedge x \in verts\ T$ **using** *to-list-tree-single* **by** *force*
**obtain** $y$ **where** $Dtree.root\ t = [y]$ **using** *0 to-list-tree-single T.adj-in-verts*(*2*) **by** *blast*
**then show** *?thesis* **using** *0 to-list-tree-def x-def*(*1*) *in-arcs-imp-in-arcs-ends* **by** *force*
**qed**

**lemma** *to-list-dtree-nempty-root*: *is-subtree* (*Node r xs*) *to-list-dtree* $\Longrightarrow r \neq []$
**using** *list-dtree.list-dtree-sub list-dtree.wf-lverts to-list-dtree-list-dtree* **by** *force*

**lemma** *dom-children-aux*:
**assumes** *is-subtree* (*Node r xs*) *to-list-dtree*
**and** *max-deg t1* $\leq$ *1*
**and** (*t1*,*e1*) $\in$ *fset xs*
**and** $x \in dlverts\ t1$
**shows** $\exists v \in set\ r \cup path\text{-}lverts\ t1\ x.\ v \rightarrow_T x$
**proof**(*cases* $x \in set\ (Dtree.root\ t1)$)
**case** *True*
**have** $Dtree.root\ t1 \in dverts\ to\text{-}list\text{-}dtree$
**using** *assms*(*1*,*3*) *dverts-subtree-subset dtree.set-sel*(*1*) **by** *fastforce*
**then have** $Dtree.root\ t1 = [x]$ **using** *to-list-dtree-single True* **by** *fastforce*
**then have** *0*: $hd\ r \rightarrow_T x$ **using** *subtree-to-list-dtree-dom assms*(*1*,*3*) **by** *fastforce*
**have** $r \in dverts\ to\text{-}list\text{-}dtree$ **using** *assms*(*1*) *dverts-subtree-subset* **by** *force*
**then have** $r = [hd\ r]$ **using** *to-list-dtree-single True* **by** *fastforce*
**then have** $hd\ r \in set\ r$ **using** *hd-in-set*[*of r*] **by** *blast*
**then show** *?thesis* **using** *0* **by** *blast*
**next**
**case** *False*
**obtain** *t2* **where** *t2-def*: *is-subtree t2 t1 x* $\in$ *set* (*Dtree.root t2*)
**using** *assms*(*4*) *subtree-root-if-dlverts* **by** *fastforce*
**then obtain** *r1 xs1* **where** *r1-def*: *is-subtree* (*Node r1 xs1*) *t1 t2* $\in$ *fst* ‘ *fset xs1*
**using** *subtree-child-if-strict-subtree t2-def False* **unfolding** *strict-subtree-def* **by** *blast*
**have** *is-subtree* (*Node r1 xs1*) (*Node r xs*) **using** *r1-def*(*1*) *assms*(*3*) **by** *auto*
**then have** *sub-r1*: *is-subtree* (*Node r1 xs1*) *to-list-dtree* **using** *assms*(*1*) *subtree-trans* **by** *blast*
**have** *sub-t1-r*: *is-subtree t1* (*Node r xs*)
**using** *subtree-if-child*[*of t1 xs*] *assms*(*3*) **by** *force*
**then have** *is-subtree t2 to-list-dtree* **using** *assms*(*1*) *subtree-trans t2-def*(*1*) **by** *blast*
**then have** $Dtree.root\ t2 \in dverts\ to\text{-}list\text{-}dtree$
**using** *assms*(*1*) *dverts-subtree-subset dtree.set-sel*(*1*) **by** *fastforce*
**then have** $Dtree.root\ t2 = [x]$ **using** *to-list-dtree-single t2-def*(*2*) **by** *force*
**then have** *0*: $hd\ r1 \rightarrow_T x$ **using** *subtree-to-list-dtree-dom*[*OF sub-r1*] *r1-def*(*2*) **by** *fastforce*

**have** *sub-t1-to*: *is-subtree t1 to-list-dtree* **using** *sub-t1-r assms(1) subtree-trans* **by** *blast*
**then have** *wf-dlverts t1* **using** *t.wf-lverts list-dtree-def t.list-dtree-sub* **by** *blast*
**moreover have** *max-deg t1* $\leq$ *1* **using** *assms(2) sub-t1-r le-trans mdeg-ge-sub* **by** *blast*
**ultimately have** *set r1* $\subseteq$ *path-lverts t1 x*
**using** *subtree-path-lverts-sub r1-def t2-def(2)* **by** *fast*
**then show** *?thesis*
**using** *0 sub-r1 dverts-subtree-subset hd-in-set[of r1] to-list-dtree-single* **by** *force*
**qed**

**lemma** *hd-dverts-in-dlverts*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; (*t1*,*e1*) $\in$ *fset xs*; *x* $\in$ *dverts t1*⟧ $\Longrightarrow$ *hd x* $\in$ *dlverts t1*
**using** *list-dtree.list-dtree-rec list-dtree.wf-lverts hd-in-lverts-if-wf t.list-dtree-sub*
**by** *fastforce*

**lemma** *dom-children-aux2*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; *max-deg t1* $\leq$ *1*; (*t1*,*e1*) $\in$ *fset xs*; *x* $\in$ *dverts t1*⟧
$\Longrightarrow$ $\exists$ *v* $\in$ *set r* $\cup$ *path-lverts t1* (*hd x*). *v* $\rightarrow_T$ (*hd x*)
**using** *dom-children-aux hd-dverts-in-dlverts* **by** *blast*

**lemma** *dom-children-full*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; $\forall$ *t* $\in$ *fst* ‘ *fset xs*. *max-deg t* $\leq$ *1*⟧
$\Longrightarrow$ *dom-children* (*Node r xs*) *T*
**unfolding** *dom-children-def* **using** *dom-children-aux2* **by** *auto*

**lemma** *dom-children′*:
**assumes** *is-subtree* (*Node r xs*) *to-list-dtree*
**shows** *dom-children* (*Node r* (*Abs-fset* (*children-deg1 xs*))) *T*
**unfolding** *dom-children-def dtree.sel children-deg1-fset-id*
**using** *dom-children-aux2*[*OF assms(1)*] **by** *fastforce*

**lemma** *dom-children-maxdeg-1*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; *max-deg* (*Node r xs*) $\leq$ *1*⟧
$\Longrightarrow$ *dom-children* (*Node r xs*) *T*
**proof** (*elim dom-children-full*)
**show** *max-deg* (*Node r xs*) $\leq$ *1* $\Longrightarrow$ $\forall$ *t*$\in$*fst* ‘ *fset xs*. *max-deg t* $\leq$ *1*
**using** *mdeg-ge-child* **by** *fastforce*
**qed**

**lemma** *dom-child-subtree*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; *t* $\in$ *fst* ‘ *fset xs*⟧ $\Longrightarrow$ $\exists$ *v*$\in$*set r*. *v* $\rightarrow_T$ *hd* (*Dtree.root t*)
**using** *subtree-to-list-dtree-dom hd-in-set to-list-dtree-nempty-root* **by** *blast*

**lemma** *dom-children-maxdeg-1-self*:
*max-deg to-list-dtree* $\leq$ *1* $\Longrightarrow$ *dom-children to-list-dtree T*

**using** *dom-children-maxdeg-1*[*of Dtree.root to-list-dtree sucs to-list-dtree*] *self-subtree*
**by** *auto*

**lemma** *seq-conform-list-tree*: $\forall\, v{\in}$*verts to-list-tree. seq-conform v*
**by** (*simp add*: *to-list-tree-def seq-conform-single*)

**lemma** *conform-list-dtree*: $\forall\, v{\in}$*dverts to-list-dtree. seq-conform v*
**using** *seq-conform-list-tree dverts-eq-verts-to-list-tree* **by** *blast*

**lemma** *to-list-dtree-vert-single*: $[\![v \in$ *dverts to-list-dtree*; $x \in$ *set* $v]\!] \Longrightarrow v = [x] \wedge x \in$ *verts* $T$
**using** *to-list-dtree-single* **by** *fastforce*

**lemma** *to-list-dtree-vert-single-sub*:
$[\![$*is-subtree* (*Node r xs*) *to-list-dtree*; $x \in$ *set* $r]\!] \Longrightarrow r = [x] \wedge x \in$ *verts* $T$
**using** *to-list-dtree-vert-single dverts-subtree-subset* **by** *fastforce*

**lemma** *to-list-dtree-child-if-to-list-tree-arc*:
$[\![$*is-subtree* (*Node r xs*) *to-list-dtree*; $r \rightarrow_{\textit{to-list-tree}} v]\!] \Longrightarrow \exists\, ys.$ (*Node v ys*) $\in$ *fst* ‘ *fset xs*
**using** *finite-directed-tree.child-if-dominated-to-dtree'*[*OF to-list-tree-finite-directed-tree*]
**unfolding** *to-list-dtree-def* **by** *simp*

**lemma** *to-list-dtree-child-if-arc*:
$[\![$*is-subtree* (*Node r xs*) *to-list-dtree*; $x \in$ *set* $r$; $x \rightarrow_T y]\!]$
$\Longrightarrow \exists\, ys.$ *Node* $[y]$ *ys* $\in$ *fst* ‘ *fset xs*
**using** *to-list-dtree-child-if-to-list-tree-arc to-list-tree-dom-iff to-list-dtree-vert-single-sub*
**by** *auto*

**lemma** *to-list-dtree-dverts-if-arc*:
$[\![$*is-subtree* (*Node r xs*) *to-list-dtree*; $x \in$ *set* $r$; $x \rightarrow_T y]\!] \Longrightarrow [y] \in$ *dverts* (*Node r xs*)
**using** *to-list-dtree-child-if-arc*[*of r xs x y*] **by** *fastforce*

**lemma** *to-list-dtree-dlverts-if-arc*:
$[\![$*is-subtree* (*Node r xs*) *to-list-dtree*; $x \in$ *set* $r$; $x \rightarrow_T y]\!] \Longrightarrow y \in$ *dlverts* (*Node r xs*)
**using** *to-list-dtree-child-if-arc*[*of r xs x y*] **by** *fastforce*

**theorem** *to-list-dtree-ranked-orig*: *ranked-dtree-with-orig to-list-dtree rank cost cmp T root*
**using** *dom-children' to-list-dtree-dlverts-if-arc asi-rank* **apply**(*unfold-locales*)
**by** (*auto simp*: *dom-children-maxdeg-1 dom-child-subtree distinct-to-list-dtree conform-list-dtree*)

**interpretation** *t*: *ranked-dtree-with-orig to-list-dtree* **by** (*rule to-list-dtree-ranked-orig*)

**lemma** *forward-ikkbz-sub*: *forward ikkbz-sub*
**using** *ikkbz-sub-def dom-children-maxdeg-1-self t.ikkbz-sub-forward* **by** *simp*

## 10.6 Optimality of IKKBZ-Sub

**lemma** *ikkbz-sub-optimal-Q*:
(∀ *as*. *fwd-sub root* (*verts to-list-tree*) *as* ⟶ *cost* (*rev ikkbz-sub*) ≤ *cost* (*rev as*))
**using** *t.denormalize-ikkbz-sub-optimal to-list-dtree-hd-root-eq-root dom-children-maxdeg-1-self*
**unfolding** *dverts-eq-verts-to-list-tree ikkbz-sub-def* **by** *blast*

**lemma** *to-list-tree-sublist-if-set-eq*:
**assumes** *set ys* = ⋃(*set* ' *verts to-list-tree*) **and** *xs* ∈ *verts to-list-tree*
**shows** *sublist xs ys*
**proof** −
**obtain** *x* **where** *x-def*: *xs* = [*x*] *x* ∈ *verts T* **using** *to-list-tree-single assms*(*2*)
**by** *blast*
**then have** *x* ∈ *set ys* **using** *assms*(*1*) *to-list-tree-def* **by** *simp*
**then show** *?thesis* **using** *x-def*(*1*) *split-list*[*of x ys*] *sublist-Cons sublist-append-leftI*
**by** *fast*
**qed**

**lemma** *hd-eq-tk1-if-set-eq-verts*: *set xs* = *verts T* ⟹ *hd xs* = *root* ⟷ *take 1 xs* = [*root*]
**using** *hd-eq-take1 take1-eq-hd*[*of xs*] *non-empty* **by** *fastforce*

**lemma** *ikkbz-sub-optimal*:
⟦*set xs* = *verts T*; *distinct xs*; *forward xs*; *hd xs* = *root*⟧
⟹ *cost* (*rev ikkbz-sub*) ≤ *cost* (*rev xs*)
**using** *ikkbz-sub-optimal-Q to-list-tree-sublist-if-set-eq*
**by** (*simp add*: *hd-eq-tk1-if-set-eq-verts to-list-tree-union-verts-eq fwd-sub-def unique-set-r-def*)

**end**

## 10.7 Optimality of IKKBZ

**context** *ikkbz-query-graph*
**begin**

Optimality only with respect to valid solutions (i.e. contain every relation exactly once). Furthermore, only join trees without cross products are considered.

**lemma** *ikkbz-sub-optimal-cost-r*:
⟦*set xs* = *verts G*; *distinct xs*; *no-cross-products* (*create-ldeep xs*); *hd xs* = *r*; *r* ∈ *verts G*⟧
⟹ *cost-r r* (*rev* (*ikkbz-sub r*)) ≤ *cost-r r* (*rev xs*)
**using** *precedence-graph.ikkbz-sub-optimal verts-dir-tree-r-eq*
**by** (*fast intro*: *forward-if-ldeep-no-cross precedence-graph-r*)

**lemma** *ikkbz-sub-no-cross*: *r* ∈ *verts G* ⟹ *no-cross-products* (*create-ldeep* (*ikkbz-sub r*))
**using** *precedence-graph.forward-ikkbz-sub ikkbz-sub-verts-eq*
**by** (*fastforce intro*: *no-cross-ldeep-if-forward′ precedence-graph-r*)

**lemma** *ikkbz-sub-cost-r-eq-cost*:
*r* ∈ *verts G* ⟹ *cost-r r* (*rev* (*ikkbz-sub r*)) = *cost-l* (*ikkbz-sub r*)
**using** *ikkbz-sub-verts-eq ikkbz-sub-distinct ikkbz-sub-no-cross ikkbz-sub-hd-eq-root*
**by** (*fastforce dest*: *cost-correct′*)

**corollary** *ikkbz-sub-optimal*:
⟦*set xs* = *verts G*; *distinct xs*; *no-cross-products* (*create-ldeep xs*); *hd xs* = *r*; *r* ∈ *verts G*⟧
⟹ *cost-l* (*ikkbz-sub r*) ≤ *cost-l xs*
**using** *ikkbz-sub-optimal-cost-r cost-correct′ ikkbz-sub-cost-r-eq-cost* **by** *fastforce*

**lemma** *ikkbz-no-cross*: *no-cross-products* (*create-ldeep ikkbz*)
**using** *ikkbz-eq-ikkbz-sub ikkbz-sub-no-cross* **by** *force*

**lemma** *hd-in-verts-if-set-eq*: *set xs* = *verts G* ⟹ *hd xs* ∈ *verts G*
**using** *verts-nempty set-empty2*[*of xs*] **by** *force*

**lemma** *ikkbz-optimal*:
⟦*set xs* = *verts G*; *distinct xs*; *no-cross-products* (*create-ldeep xs*)⟧
⟹ *cost-l ikkbz* ≤ *cost-l xs*
**using** *ikkbz-min-ikkbz-sub ikkbz-sub-optimal* **by** (*fastforce intro*: *hd-in-verts-if-set-eq*)

**theorem** *ikkbz-optimal-tree*:
⟦*valid-tree t*; *no-cross-products t*; *left-deep t*⟧ ⟹ *cost* (*create-ldeep ikkbz*) ≤ *cost t*
**using** *ikkbz-optimal inorder-eq-set* **by** (*fastforce simp*: *distinct-relations-def valid-tree-def*)

**end**

**end**

**theory** *IKKBZ-Examples*
**imports** *IKKBZ-Optimality*
**begin**

# 11 Examples of Applying IKKBZ

## 11.1 Computing Contributing Selectivity without Lists

**context** *directed-tree*
**begin**

**definition** *contr-sel* :: *′a selectivity* ⇒ *′a* ⇒ *real* **where**
*contr-sel sel y* = (*if* ∃ *x*. *x* →$_T$ *y then sel* (*THE x*. *x* →$_T$ *y*) *y else 1*)

**definition** *tree-sel* :: *′a selectivity* ⇒ *bool* **where**
*tree-sel sel* = (∀ *x y*. ¬(*x* →$_T$ *y* ∨ *y* →$_T$ *x*) ⟶ *sel x y* = *1*)

**lemma** *contr-sel-gt0*: *sel-reasonable sf* $\Longrightarrow$ *contr-sel sf x* $>$ *0*
**unfolding** *contr-sel-def sel-reasonable-def* **by** *simp*

**lemma** *contr-sel-le1*: *sel-reasonable sf* $\Longrightarrow$ *contr-sel sf x* $\leq$ *1*
**unfolding** *contr-sel-def sel-reasonable-def* **by** *simp*

**lemma** *nempty-if-not-fwd-conc*: $\neg$*forward-arcs* (*y*#*xs*) $\Longrightarrow$ *xs* $\neq$ []
**by** *auto*

**lemma** *len-gt1-if-not-fwd-conc*: $\neg$*forward-arcs* (*y*#*xs*) $\Longrightarrow$ *length* (*y*#*xs*) $>$ *1*
**by** *auto*

**lemma** *two-elems-if-not-fwd-conc*: $\neg$*forward-arcs* (*y*#*xs*) $\Longrightarrow$ $\exists$ *a b cs. a* # *b* # *cs* = *y*#*xs*
**by** (*metis forward-arcs.cases forward-arcs.simps*(*2*))

**lemma** *hd-reach-all-if-nfwd-app-fwd*:
⟦$\neg$*forward-arcs* (*y*#*xs*); *forward-arcs* (*y*#*ys*@*xs*); *x* $\in$ *set* (*y*#*ys*@*xs*)⟧
$\Longrightarrow$ *hd* (*rev* (*y*#*ys*@*xs*)) $\rightarrow^*{}_T$ *x*
**using** *hd-reach-all-forward'*[*of rev* (*y*#*ys*@*xs*)] *len-gt1-if-not-fwd-conc forward-arcs-alt*
**by** *auto*

**lemma** *hd-not-y-if-if-nfwd-app-fwd*:
**assumes** $\neg$*forward-arcs* (*y*#*xs*) **and** *forward-arcs* (*y*#*ys*@*xs*)
**shows** *hd* (*rev* (*y*#*ys*@*xs*)) $\neq$ *y*
**proof** −
**obtain** *a* **where** *a-def*: *a* $\in$ *set* (*ys*@*xs*) *a* $\rightarrow_T$ *y*
**by** (*metis assms Nil-is-append-conv forward-arcs.simps*(*3*) *neq-Nil-conv*)
**then have** *hd* (*rev* (*y*#*ys*@*xs*)) $\rightarrow^*{}_T$ *a* **using** *hd-reach-all-if-nfwd-app-fwd*[*OF assms*] **by** *simp*
**then show** *?thesis*
**using** *a-def*(*2*) *reachable1-not-reverse*
**by** (*metis loopfree.adj-not-same reachable-adjI reachable-neq-reachable1*)
**qed**

**lemma** *hd-reach1-y-if-nfwd-app-fwd*:
⟦$\neg$*forward-arcs* (*y*#*xs*); *forward-arcs* (*y*#*ys*@*xs*)⟧ $\Longrightarrow$ *hd* (*rev* (*y*#*ys*@*xs*)) $\rightarrow^+{}_T$ *y*
**using** *hd-not-y-if-if-nfwd-app-fwd hd-reach-all-if-nfwd-app-fwd* **by** *auto*

**lemma** *not-fwd-if-skip1*:
⟦$\neg$ *forward-arcs* (*y*#*x*#*x'*#*xs*); *forward-arcs* (*x*#*x'*#*xs*)⟧ $\Longrightarrow$ $\neg$ *forward-arcs* (*y*#*x'*#*xs*)
**by** *auto*

**lemma** *fwd-arcs-conc-nlast-elem*:
**assumes** *forward-arcs xs* **and** *y* $\in$ *set xs* **and** *y* $\neq$ *last xs*
**shows** *forward-arcs* (*y*#*xs*)
**proof** −

**obtain** *as bs* **where** *as-def*: *as* @ *y* # *bs* = *xs bs* ≠ []
**using** *split-list-not-last*[*OF assms*(*2*,*3*)] **by** *blast*
**then have** *forward-arcs* (*y*#*bs*) **using** *assms*(*1*) *forward-arcs-split* **by** *blast*
**then obtain** *x* **where** *x-def*: *x* ∈ *set bs x* →$_T$ *y*
**using** *as-def*(*2*) **by** (*force intro*: *list.exhaust*)
**then have** *x* ∈ *set xs* **using** *as-def*(*1*) **by** *auto*
**then show** *?thesis* **using** *assms*(*1*) *x-def*(*2*) *forward-arcs.elims*(*3*) **by** *blast*
**qed**

**lemma** *fwd-app-nhead-elem*: ⟦*forward xs*; *y* ∈ *set xs*; *y* ≠ *hd xs*⟧ ⟹ *forward* (*xs*@[*y*])
**using** *fwd-arcs-conc-nlast-elem forward-arcs-alt* **by** (*simp add*: *last-rev*)

**lemma** *hd-last-not-fwd-arcs*: ¬*forward-arcs* (*x*#*xs*@[*x*])
**proof**
**assume** *asm*: *forward-arcs* (*x*#*xs*@[*x*])
**then obtain** *y* **where** *y-def*: *y* ∈ *set* (*xs*@[*x*]) *y* →$_T$ *x*
**by** (*metis append-is-Nil-conv forward-arcs.simps*(*3*) *no-back-arcs.cases*)
**then have** *hd-in-verts*: *hd* (*rev* (*xs* @ [*x*])) ∈ *verts T* **by** *auto*
**have** *forward-arcs* (*xs*@[*x*]) **using** *asm forward-arcs-split*[*of* [*x*] *xs*@[*x*]] **by** *simp*
**then have** *x* →$^*_T$ *y* **using** *hd-reach-all-forward*[*OF hd-in-verts*] *y-def forward-arcs-alt*
**by** *simp*
**then show** *False* **using** *y-def*(*2*) *reachable1-not-reverse* **by** *auto*
**qed**

**lemma** *hd-not-fwd-arcs*: ¬*forward-arcs* (*ys*@*x*#*xs*@[*x*])
**using** *hd-last-not-fwd-arcs forward-arcs-split* **by** *blast*

**lemma** *hd-last-not-fwd*: ¬*forward* (*x*#*xs*@[*x*])
**using** *hd-last-not-fwd-arcs forward-arcs-alt* **by** *simp*

**lemma** *hd-not-fwd*: ¬*forward* (*x*#*xs*@[*x*]@*ys*)
**using** *hd-not-fwd-arcs forward-arcs-alt* **by** *simp*

**lemma** *y-not-dom-if-nfwd-app-fwd*:
⟦¬*forward-arcs* (*y*#*xs*); *forward-arcs* (*y*#*ys*@*xs*); *x* ∈ *set xs*⟧ ⟹ ¬ *x* →$_T$ *y*
**using** *forward-arcs-split*[*of y*#*ys xs*] *two-elems-if-not-fwd-conc* **by** *force*

**lemma** *not-y-dom-if-nfwd-app-fwd*:
⟦¬*forward-arcs* (*y*#*xs*); *forward-arcs* (*y*#*ys*@*xs*); *x* ∈ *set xs*⟧ ⟹ ¬ *y* →$_T$ *x*
**by** (*smt* (*verit*, *ccfv-threshold*) *append-is-Nil-conv forward-arcs-alt*′ *forward-arcs-split*
*forward-cons fwd-app-nhead-elem hd-append hd-reach1-y-if-nfwd-app-fwd*
*hd-reachable1-from-outside*′ *list.distinct*(*1*) *reachable1-not-reverse reachable-adjI*
*reachable-neq-reachable1 rev.simps*(*2*) *rev-append set-rev split-list*)

**lemma** *list-sel-aux*′*1-if-tree-sel-nfwd*:
⟦*tree-sel sel*; ¬*forward-arcs* (*y*#*xs*); *forward-arcs* (*y*#*ys*@*xs*)⟧
⟹ *list-sel-aux*′ *sel xs y* = *1*
**proof**(*induction xs arbitrary*: *ys rule*: *forward-arcs.induct*)

**case** (*2 x*)
**then show** *?case* **using** *not-y-dom-if-nfwd-app-fwd*[*OF 2*(*2*,*3*)] **by** (*auto simp*: *tree-sel-def*)
**next**
**case** (*3 x x' xs*)
**then have** *forward-arcs* (*x # x' # xs*)
**using** *forward-arcs-split*[*of y#ys x#x'#xs*] **by** *simp*
**then have** ¬ *forward-arcs* (*y # x' # xs*) **using** *not-fwd-if-skip1 3.prems*(*2*) **by** *blast*
**moreover have** *forward-arcs* (*y # (ys@*[*x*]) *@ x' # xs*) **using** *3* **by** *simp*
**ultimately have** *list-sel-aux' sel* (*x' # xs*) *y* = *1* **using** *3.IH*[*OF 3.prems*(*1*)] **by** *blast*
**then show** *?case*
**using** *3.prems*(*1*) *y-not-dom-if-nfwd-app-fwd*[*OF 3.prems*(*2*,*3*)]
*not-y-dom-if-nfwd-app-fwd*[*OF 3.prems*(*2*,*3*)]
**by** (*simp add*: *tree-sel-def*)
**qed**(*simp*)

**lemma** *contr-sel-eq-list-sel-aux'-if-tree-sel*:
⟦*tree-sel sel*; *distinct* (*y#xs*); *forward-arcs* (*y#xs*); *xs* ≠ []⟧
⟹ *contr-sel sel y* = *list-sel-aux' sel xs y*
**proof**(*induction xs rule*: *forward-arcs.induct*)
**case** (*2 x*)
**then have** *x* →$_T$ *y* **by** *simp*
**then have** (*THE x. x* →$_T$ *y*) = *x* **using** *two-in-arcs-contr* **by** *blast*
**then show** *?case* **using** ‹*x* →$_T$ *y*› **unfolding** *contr-sel-def* **by** *auto*
**next**
**case** (*3 x x' xs*)
**then show** *?case*
**proof**(*cases x* →$_T$ *y*)
**case** *True*
**then have** (*THE x. x* →$_T$ *y*) = *x* **using** *two-in-arcs-contr* **by** *blast*
**then have** *contr-sel*: *contr-sel sel y* = *sel x y* **using** *True* **unfolding** *contr-sel-def* **by** *auto*
**have** ¬*forward-arcs* (*y#x'#xs*) **using** *True 3.prems*(*2*) *two-in-arcs-contr* **by** *auto*
**then have** *list-sel-aux' sel* (*x'#xs*) *y* = *1*
**using** *list-sel-aux'1-if-tree-sel-nfwd*[*of sel y x'#xs* [*x*]] *3.prems*(*1*,*3*) **by** *auto*
**then show** *?thesis* **using** *contr-sel* **by** *simp*
**next**
**case** *False*
**have** ¬*y* →$_T$ *x*
**using** *3.prems*(*2*,*3*) *forward-arcs-alt' no-back-arc-if-fwd-dstct*
**by** (*metis distinct-rev list.set-intros*(*1*) *rev.simps*(*2*) *set-rev*)
**then have** *sel x y* = *1* **using** *3.prems*(*1*) *False* **unfolding** *tree-sel-def* **by** *blast*
**then show** *?thesis* **using** *3 False* **by** *simp*
**qed**
**qed**(*simp*)

**corollary** *contr-sel-eq-list-sel-aux′-if-tree-sel′*:
⟦*tree-sel sel*; *distinct* (*xs*@[*y*]); *forward* (*xs*@[*y*]); *xs* ≠ []⟧
⟹ *contr-sel sel y* = *list-sel-aux′ sel* (*rev xs*) *y*
**by** (*simp add*: *contr-sel-eq-list-sel-aux′-if-tree-sel forward-arcs-alt*)

**corollary** *contr-sel-eq-list-sel-aux′-if-tree-sel′′*:
⟦*tree-sel sel*; *distinct* (*xs*@[*y*]); *forward* (*xs*@[*y*]); *xs* ≠ []⟧
⟹ *contr-sel sel y* = *list-sel-aux′ sel xs y*
**by** (*simp add*: *contr-sel-eq-list-sel-aux′-if-tree-sel′ mset-x-eq-list-sel-aux′-eq*[*of rev xs*])

**lemma** *contr-sel-root*[*simp*]: *contr-sel sel root* = *1*
**by** (*auto simp*: *contr-sel-def dest*: *dominated-not-root*)

**lemma** *contr-sel-notvert*[*simp*]: *v* ∉ *verts T* ⟹ *contr-sel sel v* = *1*
**by** (*auto simp*: *contr-sel-def*)

**lemma** *hd-reach-all-forward-verts*:
⟦*forward xs*; *set xs* = *verts T*; *v* ∈ *verts T*⟧ ⟹ *hd xs* →*$^*_T$ *v*
**using** *hd-reach-all-forward list.set-sel*(*1*)[*of xs*] **by** *force*

**lemma** *hd-eq-root-if-forward-verts*: ⟦*forward xs*; *set xs* = *verts T*⟧ ⟹ *hd xs* = *root*
**using** *hd-reach-all-forward-verts root-if-all-reach* **by** *simp*

**lemma** *contr-sel-eq-ldeep-s-if-tree-dst-fwd-verts*:
**assumes** *tree-sel sel* **and** *distinct xs* **and** *forward xs* **and** *set xs* = *verts T*
**shows** *contr-sel sel y* = *ldeep-s sel* (*rev xs*) *y*
**proof** −
**have** *hd-root*: *hd xs* = *root* **using** *hd-eq-root-if-forward-verts assms*(*3*,*4*) **by** *blast*
**consider** *y* ∈ *set xs y* = *root* | *y* ∈ *set xs y* ≠ *root* | *y* ∉ *set xs* **by** *blast*
**then show** *?thesis*
**proof**(*cases*)
**case** *1*
**then show** *?thesis* **using** *hd-root ldeep-s-revhd1-if-distinct assms*(*2*) **by** *auto*
**next**
**case** *2*
**then obtain** *as bs* **where** *as-def*: *as* @ *y* # *bs* = *xs* **using** *split-list*[*of y*] **by** *fastforce*
**then have** *forward* (*as*@[*y*]) **using** *assms*(*3*) *forward-split*[*of as*@[*y*]] **by** *auto*
**moreover have** *distinct* (*as*@[*y*]) **using** *assms*(*2*) *as-def* **by** *auto*
**moreover have** *as* ≠ [] **using** *2 hd-root as-def* **by** *fastforce*
**ultimately have** *contr-sel sel y* = *list-sel-aux′ sel* (*rev as*) *y*
**using** *contr-sel-eq-list-sel-aux′-if-tree-sel′*[*OF assms*(*1*)] **by** *blast*
**then show** *?thesis* **using** *as-def distinct-ldeep-s-eq-aux′*[*of rev xs*] *assms*(*2*) **by** *auto*
**next**
**case** *3*
**then have** *contr-sel sel y* = *1* **using** *assms*(*4*) **by** *simp*
**then show** *?thesis* **using** *3 ldeep-s-1-if-nelem set-rev* **by** *fastforce*

**qed**
**qed**

**corollary** *contr-sel-eq-ldeep-s-if-tree-dst-fwd-verts'*:
⟦*tree-sel sel*; *distinct xs*; *forward xs*; *set xs* = *verts T*⟧
⟹ *contr-sel sel* = *ldeep-s sel* (*rev xs*)
**using** *contr-sel-eq-ldeep-s-if-tree-dst-fwd-verts* **by** *blast*

**lemma** *add-leaf-forward-arcs-preserv*:
⟦*a* ∉ *arcs T*; *u* ∈ *verts T*; *v* ∉ *verts T*; *forward-arcs xs*⟧
⟹ *directed-tree.forward-arcs* (|*verts* = *verts T* ∪ {*v*}, *arcs* = *arcs T* ∪ {*a*},
*tail* = (*tail T*)(*a* := *u*), *head* = (*head T*)(*a* := *v*)|) *xs*
**proof**(*induction xs rule*: *forward-arcs.induct*)
**case** *1*
**then show** *?case* **using** *directed-tree.forward-arcs.simps*(*1*) *add-leaf-dir-tree* **by** *fast*
**next**
**case** (*2 x*)
**then show** *?case* **using** *directed-tree.forward-arcs.simps*(*2*) *add-leaf-dir-tree* **by** *fast*
**next**
**case** (*3 x y xs*)
**let** *?T* = (|*verts* = *verts T* ∪ {*v*}, *arcs* = *arcs T* ∪ {*a*},
*tail* = (*tail T*)(*a* := *u*), *head* = (*head T*)(*a* := *v*)|)
**interpret** *T*: *directed-tree ?T root* **using** *add-leaf-dir-tree*[*OF 3.prems*(*1*−*3*)] **by** *blast*
**have** *T.forward-arcs* (*y* # *xs*) **using** *3* **by** *fastforce*
**then show** *?case*
**using** *T.forward-arcs.simps*(*3*)[*of x y xs*] *add-leaf-dom-preserv 3.prems*(*1*,*4*) **by** *fastforce*
**qed**

**end**

## 11.2 Contributing Selectivity Satisfies ASI Property

**context** *finite-directed-tree*
**begin**

**lemma** *dst-fwd-arcs-all-verts-ex*: ∃ *xs*. *forward-arcs xs* ∧ *distinct xs* ∧ *set xs* = *verts T*
**using** *finite-verts* **proof**(*induction rule*: *finite-directed-tree-induct*)
**case** (*single-vert t h root*)
**then show** *?case* **using** *directed-tree.forward-arcs.simps*(*2*)[*OF dir-tree-single*] **by** *fastforce*
**next**
**case** (*add-leaf T' V A t h u root a v*)
**define** *T* **where** *T* ≡ (|*verts* = *V* ∪ {*v*}, *arcs* = *A* ∪ {*a*}, *tail* = *t*(*a* := *u*), *head* = *h*(*a* := *v*)|)

**interpret** $T'$: *directed-tree* $T'$ *root* **using** *add-leaf.hyps*(*3*) **by** *blast*
**interpret** $T$: *directed-tree* $T$ *root* **using** *add-leaf.hyps*(*1*,*4*−*6*) $T'$*.add-leaf-dir-tree T-def* **by** *simp*
**obtain** *xs* **where** *xs-def*: $T'$*.forward-arcs xs distinct xs set xs* = *verts* $T'$
**using** *add-leaf.IH* **by** *blast*
**then have** $T$*.forward-arcs xs*
**using** $T'$*.add-leaf-forward-arcs-preserv add-leaf.hyps*(*1*,*4*,*5*,*6*) *T-def* **by** *simp*
**moreover have** ∃ *y*∈*set xs*. *y* →$_T$ *v*
**using** *add-leaf.hyps*(*1*,*4*) *T-def xs-def*(*3*) **unfolding** *arcs-ends-def arc-to-ends-def* **by** *force*
**ultimately have** $T$*.forward-arcs* (*v*#*xs*) **using** $T$*.forward-arcs.elims*(*3*) **by** *blast*
**then show** *?case* **using** *xs-def*(*2*,*3*) *add-leaf.hyps*(*1*,*5*) *T-def* **by** *auto*
**qed**

**lemma** *dst-fwd-all-verts-ex*: ∃ *xs*. *forward xs* ∧ *distinct xs* ∧ *set xs* = *verts T*
**using** *dst-fwd-arcs-all-verts-ex forward-arcs-alt'*[*symmetric*] **by** *auto*

**lemma** *c-list-asi-if-tree-sel*:
**fixes** *sf cf h r*
**defines** *rank* ≡ (λ*l*. (*ldeep-T* (*contr-sel sf*) *cf l* − *1*) / *c-list* (*contr-sel sf*) *cf h r l*)
**assumes** *tree-sel sf*
**and** *sel-reasonable sf*
**and** ∀ *x*. *cf x* > *0*
**and** ∀ *x*. *h x* > *0*
**shows** *asi rank r* (*c-list* (*contr-sel sf*) *cf h r*)
**using** *c-list-asi assms contr-sel-eq-ldeep-s-if-tree-dst-fwd-verts' dst-fwd-all-verts-ex*
**by** *fastforce*

**end**

**context** *tree-query-graph*
**begin**

**abbreviation** *sel-r* :: *'a* ⇒ *'a* ⇒ *real* **where**
*sel-r r* ≡ *directed-tree.contr-sel* (*dir-tree-r r*) *match-sel*

Since cf is only required to be positive for verts of G, we map all others to 1.

**definition** *cf'* :: *'a* ⇒ *real* **where**
*cf' x* = (*if x* ∈ *verts G then cf x else 1*)

**definition** *c-list-r* :: (*'a* ⇒ *real*) ⇒ *'a* ⇒ *'a list* ⇒ *real* **where**
*c-list-r h r* = *c-list* (*sel-r r*) *cf' h r*

**definition** *rank-r* :: (*'a* ⇒ *real*) ⇒ *'a* ⇒ *'a list* ⇒ *real* **where**
*rank-r h r xs* = (*ldeep-T* (*sel-r r*) *cf' xs* − *1*) / *c-list-r h r xs*

**lemma** *dom-in-dir-tree-r*:

**assumes** $r \in verts\ G$ **and** $x \rightarrow_G y$
**shows** $x \rightarrow_{dir\text{-}tree\text{-}r\ r} y \vee y \rightarrow_{dir\text{-}tree\text{-}r\ r} x$
**proof** −
**obtain** *e1* **where** *e1-def*: $e1 \in arcs\ G\ tail\ G\ e1 = x\ head\ G\ e1 = y$
**using** *assms*(*2*) **unfolding** *arcs-ends-def arc-to-ends-def* **by** *blast*
**then show** *?thesis*
**proof**(*cases* $e1 \in arcs\ (dir\text{-}tree\text{-}r\ r)$)
**case** *True*
**moreover have** $tail\ (dir\text{-}tree\text{-}r\ r)\ e1 = x$
**using** *e1-def*(*2*) *tail-dir-tree-r-eq*[*OF assms*(*1*)] **by** *blast*
**moreover have** $head\ (dir\text{-}tree\text{-}r\ r)\ e1 = y$
**using** *e1-def*(*3*) *head-dir-tree-r-eq*[*OF assms*(*1*)] **by** *blast*
**ultimately show** *?thesis* **using** *e1-def*(*1*) **unfolding** *arcs-ends-def arc-to-ends-def*
**by** *blast*
**next**
**case** *False*
**then obtain** *e2* **where** *e2-def*: $e2 \in arcs\ (dir\text{-}tree\text{-}r\ r)\ tail\ G\ e2 = y\ head\ G\ e2 = x$
**using** *arcs-compl-un-eq-arcs*[*OF assms*(*1*)] *e1-def* **by** *force*
**have** $tail\ (dir\text{-}tree\text{-}r\ r)\ e2 = y$
**using** *e2-def*(*2*) *tail-dir-tree-r-eq*[*OF assms*(*1*)] **by** *blast*
**moreover have** $head\ (dir\text{-}tree\text{-}r\ r)\ e2 = x$
**using** *e2-def*(*3*) *head-dir-tree-r-eq*[*OF assms*(*1*)] **by** *blast*
**ultimately show** *?thesis* **using** *e2-def*(*1*) **unfolding** *arcs-ends-def arc-to-ends-def*
**by** *blast*
**qed**
**qed**

**lemma** *dom-in-dir-tree-r-iff-aux*:
$r \in verts\ G \Longrightarrow (x \rightarrow_{dir\text{-}tree\text{-}r\ r} y \vee y \rightarrow_{dir\text{-}tree\text{-}r\ r} x) \longleftrightarrow (x \rightarrow_G y \vee y \rightarrow_G x)$
**using** *dir-tree-r-dom-in-G dom-in-dir-tree-r* **by** *blast*

**lemma** *dom-in-dir-tree-r-iff*:
$r \in verts\ G \Longrightarrow (x \rightarrow_{dir\text{-}tree\text{-}r\ r} y \vee y \rightarrow_{dir\text{-}tree\text{-}r\ r} x) \longleftrightarrow x \rightarrow_G y$
**using** *dom-in-dir-tree-r-iff-aux dominates-sym* **by** *blast*

**lemma** *dir-tree-sel*[*intro*]: $r \in verts\ G \Longrightarrow directed\text{-}tree.tree\text{-}sel\ (dir\text{-}tree\text{-}r\ r)\ match\text{-}sel$
**unfolding** *directed-tree.tree-sel-def*[*OF directed-tree-r*]
**using** *match-sel1-if-no-arc dom-in-dir-tree-r-iff* **by** *blast*

**lemma** *pos-cards'*[*intro*!]: $\forall x.\ cf'\ x > 0$
**unfolding** *cf'-def* **using** *pos-cards* **by** *simp*

**theorem** *c-list-asi*: $[\![r \in verts\ G; \forall x.\ h\ x > 0]\!] \Longrightarrow asi\ (rank\text{-}r\ h\ r)\ r\ (c\text{-}list\text{-}r\ h\ r)$
**using** *finite-directed-tree.c-list-asi-if-tree-sel*[*OF fin-directed-tree-r*]
**unfolding** *c-list-r-def rank-r-def* **by** *blast*

## 11.3 Applying IKKBZ

**lemma** *cf'-simp*: $x \in verts\ G \Longrightarrow cf'\ x = cf\ x$
**unfolding** *cf'-def* **by** *simp*

**lemma** *ldeep-T-cf'-eq*: $set\ xs \subseteq verts\ G \Longrightarrow ldeep\text{-}T\ sf\ cf'\ xs = ldeep\text{-}T\ sf\ cf\ xs$
**using** *ldeep-T-eq-if-cf-eq*[*of xs*] *cf'-simp* **by** *blast*

**lemma** *clist-cf'-eq*: $set\ xs \subseteq verts\ G \Longrightarrow c\text{-}list\ sf\ cf'\ h\ r\ xs = c\text{-}list\ sf\ cf\ h\ r\ xs$
**by** (*simp add*: *clist-eq-if-cf-eq ldeep-T-cf'-eq*)

**lemma** *card-cf'-eq*: $matching\text{-}rels\ t \Longrightarrow card\ cf'\ f\ t = card\ cf\ f\ t$
**by** (*induction cf' f t rule*: *card.induct*) (*auto simp*: *matching-rels-def cf'-simp*)

**lemma** *c-IKKBZ-cf'-eq*: $matching\text{-}rels\ t \Longrightarrow c\text{-}IKKBZ\ h\ cf'\ sf\ t = c\text{-}IKKBZ\ h\ cf\ sf\ t$
**by** (*induction h cf' sf t rule*: *c-IKKBZ.induct*) (*auto simp*: *card-cf'-eq cf'-simp matching-rels-def*)

**lemma** *c-IKKBZ-cf'-eq'*: $valid\text{-}tree\ t \Longrightarrow c\text{-}IKKBZ\ h\ cf'\ sf\ t = c\text{-}IKKBZ\ h\ cf\ sf\ t$
**by** (*simp add*: *c-IKKBZ-cf'-eq matching-rels-def valid-tree-def*)

**lemma** *c-out-cf'-eq*: $matching\text{-}rels\ t \Longrightarrow c\text{-}out\ cf'\ sf\ t = c\text{-}out\ cf\ sf\ t$
**by** (*induction cf' sf t rule*: *c-out.induct*) (*auto simp*: *card-cf'-eq cf'-simp matching-rels-def*)

**lemma** *c-out-cf'-eq'*: $valid\text{-}tree\ t \Longrightarrow c\text{-}out\ cf'\ sf\ t = c\text{-}out\ cf\ sf\ t$
**by** (*simp add*: *c-out-cf'-eq matching-rels-def valid-tree-def*)

**lemma** *joinTree-card'-pos*[*intro*]: *pos-rel-cards cf' t*
**by** (*induction t*) (*auto simp*: *pos-cards' pos-rel-cards-def*)

**lemma** *match-reasonable-cards'*[*intro*]: *reasonable-cards cf' match-sel t*
**using** *pos-sel-reason-impl-reason* **by** *blast*

**lemma** *sel-r-gt0*: $r \in verts\ G \Longrightarrow sel\text{-}r\ r\ x > 0$
**using** *directed-tree.contr-sel-gt0*[*OF directed-tree-r*] **by** *blast*

**lemma** *sel-r-le1*: $r \in verts\ G \Longrightarrow sel\text{-}r\ r\ x \leq 1$
**using** *directed-tree.contr-sel-le1*[*OF directed-tree-r*] **by** *blast*

**lemma** *sel-r-eq-ldeep-s-if-dst-fwd-verts*:
$[\![r \in verts\ G;\ distinct\ xs;\ directed\text{-}tree.forward\ (dir\text{-}tree\text{-}r\ r)\ xs;\ set\ xs = verts\ G]\!]$
$\Longrightarrow sel\text{-}r\ r = ldeep\text{-}s\ match\text{-}sel\ (rev\ xs)$
**using** *directed-tree.contr-sel-eq-ldeep-s-if-tree-dst-fwd-verts'*[*OF directed-tree-r*]
*verts-dir-tree-r-eq*
**by** *blast*

**lemma** *sel-r-eq-ldeep-s-if-valid-fwd*:
$[\![r \in verts\ G;\ valid\text{-}tree\ t;\ directed\text{-}tree.forward\ (dir\text{-}tree\text{-}r\ r)\ (inorder\ t)]\!]$

$\Longrightarrow$ *sel-r r = ldeep-s match-sel (reorder t)*
**unfolding** *valid-tree-def distinct-relations-def inorder-eq-set*[*symmetric*] *reorder-eq-rev-inorder*
**using** *sel-r-eq-ldeep-s-if-dst-fwd-verts* **by** *blast*

**lemma** *sel-r-eq-ldeep-s-if-valid-no-cross*:
⟦*valid-tree t*; *no-cross-products t*; *left-deep t*⟧
$\Longrightarrow$ *sel-r* (*first-node t*) = *ldeep-s match-sel* (*reorder t*)
**using** *sel-r-eq-ldeep-s-if-valid-fwd forward-if-ldeep-no-cross′*
*valid-tree-def first-node-in-verts-if-valid*
**by** *blast*

**lemma** *c-list-ldeep-s-eq-c-list-r-if-valid-no-cross*:
⟦*valid-tree t*; *no-cross-products t*; *left-deep t*⟧
$\Longrightarrow$ *c-list* (*ldeep-s match-sel* (*reorder t*)) *cf′ h* (*first-node t*) *xs*
= *c-list-r h* (*first-node t*) *xs*
**using** *sel-r-eq-ldeep-s-if-valid-no-cross c-list-r-def* **by** *simp*

**lemma** *c-IKKBZ-list-correct-if-simple-h*:
**assumes** *valid-tree t* **and** *no-cross-products t* **and** *left-deep t*
**shows** *c-list-r* (*λx. h x* (*cf′ x*)) (*first-node t*) (*reorder t*) = *c-IKKBZ h cf match-sel t*
**proof** −
**have** (*λt. c-IKKBZ h cf′ match-sel t*) *t*
= *c-list* (*ldeep-s match-sel* (*reorder t*)) *cf′* (*λx. h x* (*cf′ x*)) (*first-node t*) (*reorder t*)
**using** *c-IKKBZ-eq-c-list assms*(*1*,*3*) *valid-tree-def* **by** *fast*
**then show** *?thesis*
**using** *c-list-ldeep-s-eq-c-list-r-if-valid-no-cross assms* **by** (*simp add*: *c-IKKBZ-cf′-eq′*)
**qed**

**end**

### 11.3.1 Applying IKKBZ on Simple Cost Functions

For simple cost functions like *c-nlj* and *c-hj* that do not depend on the contributing selectivies as *c-out* does, the h function does not change. Therefore, we can apply it directly using *c-IKKBZ* and *c-list*.

**context** *cmp-tree-query-graph*
**begin**

**context**
**fixes** *h* :: *′a* ⇒ *real* ⇒ *real*
**assumes** *h-pos*: ∀ *x. h x* (*cf′ x*) > *0*
**begin**

**theorem** *ikkbz-query-graph-if-simple-h*:
**defines** *cost* ≡ *c-IKKBZ h cf match-sel*
**defines** *h′* ≡ (*λx. h x* (*cf′ x*))
**shows** *ikkbz-query-graph bfs sel cf G cmp cost* (*c-list-r h′*) (*rank-r h′*)

**unfolding** *ikkbz-query-graph-def ikkbz-query-graph-axioms-def assms*
**by** (*auto simp*: *cmp-tree-query-graph-axioms c-list-asi c-IKKBZ-list-correct-if-simple-h h-pos*)

**interpretation** *ikkbz-query-graph bfs sel cf G cmp*
*c-IKKBZ h cf match-sel c-list-r* (λ*x*. *h x* (*cf*′ *x*)) *rank-r* (λ*x*. *h x* (*cf*′ *x*))
**by** (*fact ikkbz-query-graph-if-simple-h*)

**corollary** *ikkbz-simple-h-nempty*: *ikkbz* ≠ []
**by** (*rule ikkbz-nempty*)

**corollary** *ikkbz-simple-h-valid-tree*: *valid-tree* (*create-ldeep ikkbz*)
**by** (*rule ikkbz-valid-tree*)

**corollary** *ikkbz-simple-h-no-cross*:
*no-cross-products* (*create-ldeep ikkbz*)
**by** (*rule ikkbz-no-cross*)

**theorem** *ikkbz-simple-h-optimal*:
⟦*valid-tree t*; *no-cross-products t*; *left-deep t*⟧
⟹ *c-IKKBZ h cf match-sel* (*create-ldeep ikkbz*) ≤ *c-IKKBZ h cf match-sel t*
**by** (*rule ikkbz-optimal-tree*)

**abbreviation** *ikkbz-simple-h* :: ′*a list* **where**
*ikkbz-simple-h* ≡ *ikkbz*
**end**

We can now apply these results directly to valid cost functions like *c-nlj* and *c-hj*.

**lemma** *id-cf*′*-gt0*: ∀ *x*. *id* (*cf*′ *x*) > *0*
**by** *auto*

**corollary** *ikkbz-nempty-nlj*: *ikkbz-simple-h* (λ-. *id*) ≠ []
**using** *ikkbz-simple-h-nempty*[*of* λ-. *id*, *OF id-cf*′*-gt0*] **by** *blast*

**corollary** *ikkbz-valid-tree-nlj*: *valid-tree* (*create-ldeep* (*ikkbz-simple-h* (λ-. *id*)))
**using** *ikkbz-simple-h-valid-tree*[*of* λ-. *id*, *OF id-cf*′*-gt0*] **by** *blast*

**corollary** *ikkbz-no-cross-nlj*: *no-cross-products* (*create-ldeep* (*ikkbz-simple-h* (λ-. *id*)))
**using** *ikkbz-simple-h-no-cross*[*of* λ-. *id*, *OF id-cf*′*-gt0*] **by** *blast*

**corollary** *ikkbz-optimal-nlj*:
⟦*valid-tree t*; *no-cross-products t*; *left-deep t*⟧
⟹ *c-nlj cf match-sel* (*create-ldeep* (*ikkbz-simple-h* (λ-. *id*))) ≤ *c-nlj cf match-sel t*
**using** *ikkbz-simple-h-optimal*[*of* λ-. *id*, *OF id-cf*′*-gt0*] *ikkbz-nempty-nlj*
**by** (*fastforce simp*: *c-nlj-IKKBZ create-ldeep-ldeep*)

**corollary** *ikkbz-nempty-hj*: *ikkbz-simple-h* (λ- -. *1.2*) ≠ []
**using** *ikkbz-simple-h-nempty* **by** *force*

**corollary** *ikkbz-valid-tree-hj*: *valid-tree* (*create-ldeep* (*ikkbz-simple-h* (λ- -. *1.2*)))
**using** *ikkbz-simple-h-valid-tree* **by** *force*

**corollary** *ikkbz-no-cross-hj*: *no-cross-products* (*create-ldeep* (*ikkbz-simple-h* (λ- -. *1.2*)))
**using** *ikkbz-simple-h-no-cross* **by** *force*

**corollary** *ikkbz-optimal-hj*:
⟦*valid-tree t*; *no-cross-products t*; *left-deep t*⟧
⟹ *c-hj cf match-sel* (*create-ldeep* (*ikkbz-simple-h* (λ- -. *1.2*))) ≤ *c-hj cf match-sel t*
**using** *ikkbz-simple-h-optimal*[*of* λ- -. *1.2*] *ikkbz-nempty-hj*
**by** (*fastforce simp*: *c-hj-IKKBZ create-ldeep-ldeep*)

**end**

### 11.3.2 Applying IKKBZ on C_out

Since *c-out* uses the contributing selectivity as part of its h, we can not use the general approach we used for the "simple" cost functions. Instead, we show the applicability directly.

**context** *tree-query-graph*
**begin**

**definition** *c-out-list-r* :: *'a* ⇒ *'a list* ⇒ *real* **where**
*c-out-list-r r* = *c-list-r* (λ*a*. *sel-r r a* ∗ *cf' a*) *r*

**definition** *c-out-rank-r* :: *'a* ⇒ *'a list* ⇒ *real* **where**
*c-out-rank-r r* = *rank-r* (λ*a*. *sel-r r a* ∗ *cf' a*) *r*

**lemma** *c-out-eq-c-list-cf'*:
**fixes** *t*
**defines** *xs* ≡ *revorder t*
**defines** *h* ≡ (λ*a*. *ldeep-s match-sel xs a* ∗ *cf' a*)
**assumes** *distinct-relations t* **and** *left-deep t*
**shows** *c-list* (*ldeep-s match-sel xs*) *cf' h* (*first-node t*) *xs* = *c-out cf' match-sel t*
**using** *c-out-eq-c-list assms* **by** *blast*

**lemma** *c-out-list-correct-cf'*:
**fixes** *t*
**defines** *h* ≡ (λ*a*. *sel-r* (*first-node t*) *a* ∗ *cf' a*)
**assumes** *valid-tree t* **and** *no-cross-products t* **and** *left-deep t*
**shows** *c-list-r h* (*first-node t*) (*revorder t*) = *c-out cf' match-sel t*
**using** *c-out-eq-c-list-cf' assms sel-r-eq-ldeep-s-if-valid-no-cross*
**by** (*fastforce simp*: *valid-tree-def c-list-ldeep-s-eq-c-list-r-if-valid-no-cross*)

**lemma** *c-out-list-correct-cf*:
**fixes** *t*
**defines** $h \equiv (\lambda a.\ sel\text{-}r\ (first\text{-}node\ t)\ a * cf'\ a)$
**assumes** *valid-tree t* **and** *no-cross-products t* **and** *left-deep t*
**shows** *c-list-r h* (*first-node t*) (*revorder t*) = *c-out cf match-sel t*
**using** *c-out-list-correct-cf' c-out-cf'-eq' assms* **by** *simp*

**lemma** *c-out-list-correct*:
⟦*valid-tree t*; *no-cross-products t*; *left-deep t*⟧
⟹ *c-out-list-r* (*first-node t*) (*revorder t*) = *c-out cf match-sel t*
**using** *c-out-list-correct-cf c-out-list-r-def* **by** *simp*

**lemma** *c-out-h-gt0*: $r \in verts\ G \Longrightarrow (\lambda a.\ sel\text{-}r\ r\ a * cf'\ a)\ x > 0$
**using** *sel-r-gt0* **by** (*simp add*: *pos-cards'*)

**lemma** *c-out-r-asi*: $r \in verts\ G \Longrightarrow asi$ (*c-out-rank-r r*) *r* (*c-out-list-r r*)
**using** *c-out-h-gt0* **by** (*simp add*: *c-list-asi c-out-list-r-def c-out-rank-r-def*)

**end**

**context** *cmp-tree-query-graph*
**begin**

**theorem** *ikkbz-query-graph-c-out*:
*ikkbz-query-graph bfs sel cf G cmp* (*c-out cf match-sel*) *c-out-list-r c-out-rank-r*
**unfolding** *ikkbz-query-graph-def ikkbz-query-graph-axioms-def*
**by** (*auto simp*: *cmp-tree-query-graph-axioms c-out-r-asi c-out-list-correct*)

**interpretation** $QG_{out}$:
*ikkbz-query-graph bfs sel cf G cmp c-out cf match-sel c-out-list-r c-out-rank-r*
**by** (*rule ikkbz-query-graph-c-out*)

**corollary** *ikkbz-nempty-cout*: $QG_{out}.ikkbz \neq []$
**using** $QG_{out}.ikkbz\text{-}nempty$ **.**

**corollary** *ikkbz-valid-tree-cout*: *valid-tree* (*create-ldeep* $QG_{out}.ikkbz$)
**using** $QG_{out}.ikkbz\text{-}valid\text{-}tree$ **.**

**corollary** *ikkbz-no-cross-cout*: *no-cross-products* (*create-ldeep* $QG_{out}.ikkbz$)
**using** $QG_{out}.ikkbz\text{-}no\text{-}cross$ **.**

**corollary** *ikkbz-optimal-cout*:
⟦*valid-tree t*; *no-cross-products t*; *left-deep t*⟧
⟹ *c-out cf match-sel* (*create-ldeep* $QG_{out}.ikkbz$) ≤ *c-out cf match-sel t*
**using** $QG_{out}.ikkbz\text{-}optimal\text{-}tree$ **.**

**end**

## 11.4 Instantiating Comparators with Linorders

**locale** *alin-tree-query-graph = tree-query-graph bfs sel cf G*
  **for** *bfs sel* **and** *cf* :: *'a* :: *linorder* ⇒ *real* **and** *G*
**begin**

**lift-definition** *cmp* :: (*'a list*×*'b*) *comparator* **is**
  (λ*x y*. *if hd* (*fst x*) < *hd* (*fst y*) *then Less*
    *else if hd* (*fst x*) > *hd* (*fst y*) *then Greater else Equiv*)
  **by**(*unfold-locales*) (*auto split*: *if-splits*)

**lemma** *cmp-hd-eq-if-equiv*: *compare cmp* (*v1*,*e1*) (*v2*,*e2*) = *Equiv* ⟹ *hd v1* = *hd v2*
  **by**(*auto simp*: *cmp.rep-eq split*: *if-splits*)

**lemma** *cmp-sets-not-dsjnt-if-equiv*:
  ⟦*v1* ≠ []; *v2* ≠ []; *compare cmp* (*v1*,*e1*) (*v2*,*e2*) = *Equiv*⟧ ⟹ *set v1* ∩ *set v2* ≠ {}
  **using** *cmp-hd-eq-if-equiv disjoint-iff-not-equal hd-in-set*[*of v1*] **by** *auto*

**lemma** *cmp-tree-qg*: *cmp-tree-query-graph bfs sel cf G cmp*
  **by** *standard* (*simp add*: *cmp-sets-not-dsjnt-if-equiv*)

**interpretation** *cmp-tree-query-graph bfs sel cf G cmp*
  **by** (*rule cmp-tree-qg*)

**thm** *ikkbz-optimal-hj ikkbz-optimal-cout*

**end**

**locale** *blin-tree-query-graph = tree-query-graph bfs sel cf G*
  **for** *bfs* **and** *sel* :: *'b* :: *linorder* ⇒ *real* **and** *cf G*
**begin**

**lift-definition** *cmp* :: (*'a list*×*'b*) *comparator* **is**
  (λ*x y*. *if snd x* < *snd y then Less*
    *else if snd x* > *snd y then Greater else Equiv*)
  **by**(*unfold-locales*) (*auto split*: *if-splits*)

**lemma** *cmp-arcs-eq-if-equiv*: *compare cmp* (*v1*,*e1*) (*v2*,*e2*) = *Equiv* ⟹ *e1* = *e2*
  **by**(*auto simp*: *cmp.rep-eq split*: *if-splits*)

**lemma** *cmp-tree-qg*: *cmp-tree-query-graph bfs sel cf G cmp*
  **by** *standard* (*simp add*: *cmp-arcs-eq-if-equiv*)

**interpretation** *cmp-tree-query-graph bfs sel cf G cmp*
  **by** (*rule cmp-tree-qg*)

**thm** *ikkbz-optimal-hj ikkbz-optimal-cout*

**end**

**end**

## References


[1] C. Ballarin. Tutorial to locales and locale interpretation.

[2] S. Chaudhuri. An overview of query optimization in relational systems. In A. O. Mendelzon and J. Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43. ACM Press, 1998.

[3] W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors. *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*. Morgan Kaufmann, 1986.

[4] L. Fegaras. A new heuristic for optimizing large queries. In G. Quirchmayr, E. Schweighofer, and T. J. M. Bench-Capon, editors, *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings*, volume 1460 of *Lecture Notes in Computer Science*, pages 726–735. Springer, 1998.

[5] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[6] A. Krauss. Defining recursive functions in isabelle/hol.

[7] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In Chu et al. [3], pages 128–137.

[8] G. Moerkotte. Building query compilers, 2020.

[9] T. Neumann and B. Radke. Query optimization lecture.

[10] T. Neumann and B. Radke. Query optimization lecture - chapter 3.

[11] T. Nipkow. Programming and proving in isabelle/hol, 2021.

[12] L. Noschinski. Graph theory. *Archive of Formal Proofs*, Apr. 2013. https://isa-afp.org/entries/Graph_Theory.html, Formal proof development.

[13] L. C. Paulson. *Isabelle - A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[14] L. Stevens and M. Abdulaziz. Fast diameter estimation.