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 **10**
- 2.1 Functions . . . 10
  - 2.1.1 Functions for Information Retrieval . . . 10
  - 2.1.2 Functions for Correctness Checks . . . 10
  - 2.1.3 Functions for Modifications . . . 11
  - 2.1.4 Additional properties . . . 11
  - 2.1.5 Cardinality Calculations for Left-deep Trees . . . 12
- 2.2 Proofs . . . 12

3 Cost Functions **20**
- 3.1 General Cost Functions . . . 20
- 3.2 Cost functions that are considered by IKKBZ. . . . 21
- 3.3 Properties of Cost Functions . . . 21
- 3.4 Proofs . . . 22
  - 3.4.1 Equivalence Proofs . . . 22
  - 3.4.2 Additional ASI Proofs . . . 32

4 Graph Extensions **34**
- 4.1 Vertices with Multiple Outgoing Arcs . . . 38
- 4.2 Vertices with Multiple Incoming Arcs . . . 41

5 Query Graphs **42**
5.1 Function for Join Trees and Selectivities . . . 42
5.2 Proofs . . . 42
5.3 Pair Query Graph . . . 46

6 Directed Tree Additions **47**
6.1 Directed Trees of Connected Trees . . . 48
6.1.1 Tranformation using BFS . . . 48
6.1.2 Tranformation using PSP-Trees . . . 53
6.2 Additions for Induction on Directed Trees . . . 59
6.3 Branching Points in Directed Trees . . . 62
6.4 Converting to Trees of Lists . . . 63

7 Algebraic Type for Directed Trees **65**
7.1 Termination Proofs . . . 65
7.2 Dtree Basic Functions . . . 65
7.3 Dtree Basic Proofs . . . 66
7.3.1 Finite Directed Trees to Dtree . . . 82
7.3.2 Well-Formed Dtrees . . . 88
7.3.3 Identity of Transformation Operations . . . 92
7.4 Degrees of Nodes . . . 94
7.5 List Conversions . . . 99
7.6 Inserting in Dtrees . . . 104

8 Dtrees of Lists **110**
8.1 Functions . . . 110
8.2 List Dtrees as Well-Formed Dtrees . . . 111
8.3 Combining Preserves Well-Formedness . . . 116

9 IKKBZ **119**
9.1 Additional Proofs for Merging Lists . . . 119
9.2 Merging Subtrees of Ranked Dtrees . . . 120
9.2.1 Definitions . . . 121
9.2.2 Commutativity Proofs . . . 121
9.2.3 Merging Preserves Arcs and Verts . . . 125
9.2.4 Merging Preserves Well-Formedness . . . 129
9.2.5 Additional Merging Properties . . . 130
9.3 Normalizing Dtrees . . . 132
9.3.1 Definitions . . . 132
9.3.2 Basic Proofs . . . 133
9.3.3 Normalizing Preserves Well-Formedness . . . 134
9.3.4 Distinctness and hd preserved . . . 135
9.3.5 Normalize and Sorting . . . 136
9.4 Removing Wedges . . . 139

9.5 IKKBZ-Sub 142
9.6 Full IKKBZ 146

10 Optimality of IKKBZ **149**
10.1 Sublist Additions 157
10.2 Optimal Solution for Lists of Fixed Sets 160
10.3 Arc Invariants 180
10.3.1 Normalizing preserves Arc Invariants 190
10.3.2 Merging preserves Arc Invariants 195
10.3.3 Merge1 preserves Arc Invariants 198
10.4 Optimality of IKKBZ-Sub result constrained to Invariants . . 201
10.4.1 Result fulfills the requirements 201
10.4.2 Minimal Cost of the result 207
10.5 Arc Invariants hold for Conversion to Dtree 213
10.6 Optimality of IKKBZ-Sub 215
10.7 Optimality of IKKBZ 215

11 Examples of Applying IKKBZ **216**
11.1 Computing Contributing Selectivity without Lists 216
11.2 Contributing Selectivity Satisfies ASI Property 219
11.3 Applying IKKBZ 220
11.3.1 Applying IKKBZ on Simple Cost Functions 222
11.3.2 Applying IKKBZ on C_out 223
11.4 Instantiating Comparators with Linorders 225

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*
  ⟨*proof*⟩

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*⟩

lemma *mset-y-eq-list-sel-eq*: *mset y* = *mset y'* ⟹ *list-sel f x y* = *list-sel f x y'*
  ⟨*proof*⟩

lemma *mset-x-eq-list-sel-eq*: *mset x* = *mset z* ⟹ *list-sel f x y* = *list-sel f z y*
⟨*proof*⟩

lemma *list-sel-empty*: *list-sel f x* [] = *1*
  ⟨*proof*⟩

lemma *list-sel'-empty*: *list-sel' f* [] *y* = *1*
  ⟨*proof*⟩

lemma *list-sel-symm-app*:
sel-symm f $\Longrightarrow$ *list-sel-aux f x y* $*$ *list-sel f y xs* $=$ *list-sel f y* (*x* # *xs*)
⟨*proof*⟩

lemma *list-sel-symm*: *sel-symm f* $\Longrightarrow$ *list-sel f x y* $=$ *list-sel f y x*
⟨*proof*⟩

lemma *list-sel-symm-aux-eq'*: *sel-symm f* $\Longrightarrow$ *list-sel-aux f x y* $=$ *list-sel-aux' f y x*
⟨*proof*⟩

lemma *list-sel-sing-aux'*: *list-sel f x* [*y*] $=$ *list-sel-aux' f x y*
⟨*proof*⟩

lemma *list-sel-sing-aux*: *list-sel f* [*x*] *y* $=$ *list-sel-aux f x y*
⟨*proof*⟩

lemma *list-sel'-sing-aux'*: *list-sel' f x* [*y*] $=$ *list-sel-aux' f x y*
⟨*proof*⟩

lemma *list-sel'-sing-aux*: *list-sel' f* [*x*] *y* $=$ *list-sel-aux f x y*
⟨*proof*⟩

lemma *list-sel'-split-aux*: *list-sel' f* (*x*#*xs*) *y* $=$ *list-sel-aux f x y* $*$ *list-sel' f xs y*
⟨*proof*⟩

lemma *list-sel-eq'*: *list-sel f x y* $=$ *list-sel' f x y*
⟨*proof*⟩

lemma *mset-x-eq-list-sel-aux'-eq*: *mset x* $=$ *mset z* $\Longrightarrow$ *list-sel-aux' f x y* $=$ *list-sel-aux' f z y*
⟨*proof*⟩

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*⟩

lemma *list-sel-aux-eq-foldl*: *list-sel-aux f x y* $=$ *foldl* (λ*a b*. *a* $*$ *f x b*) *1 y*
⟨*proof*⟩

lemma *list-sel-eq-foldl*: *list-sel f x y* $=$ *foldl* (λ*a b*. *a* $*$ *list-sel-aux f b y*) *1 x*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list-sel-aux-eq-foldr*: *list-sel-aux f x y* $=$ *foldr* (λ*b a*. *a* $*$ *f x b*) *y 1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list-sel-eq-foldr*: *list-sel f x y* = *foldr* (λ*b a*. *a* ∗ *list-sel-aux f b y*) *x 1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list-sel-aux-reasonable*:
sel-reasonable f ⟹ *list-sel-aux f x y* ≤ *1* ∧ *list-sel-aux f x y* > *0*
⟨*proof*⟩

lemma *list-sel-aux′-reasonable*:
sel-reasonable f ⟹ *list-sel-aux′ f x y* ≤ *1* ∧ *list-sel-aux′ f x y* > *0*
⟨*proof*⟩

lemma *list-sel-reasonable*: *sel-reasonable f* ⟹ *list-sel f x y* ≤ *1* ∧ *list-sel f x y* > *0*
⟨*proof*⟩

lemma *list-sel′-reasonable*: *sel-reasonable f* ⟹ *list-sel′ f x y* ≤ *1* ∧ *list-sel′ f x y* > *0*
⟨*proof*⟩

lemma *list-sel-aux-eq-set-sel-aux*:
distinct ys ⟹ *list-sel-aux f x ys* = *set-sel-aux f x* (*set ys*)
⟨*proof*⟩

lemma *list-sel-eq-set-sel*:
⟦*distinct xs*; *distinct ys*⟧ ⟹ *list-sel f xs ys* = *set-sel f* (*set xs*) (*set ys*)
⟨*proof*⟩

lemma *list-sel′-eq-set-sel*:
⟦*distinct xs*; *distinct ys*⟧ ⟹ *list-sel′ f xs ys* = *set-sel f* (*set xs*) (*set ys*)
⟨*proof*⟩

lemma *set-sel-symm-if-finite*: ⟦*finite X*; *finite Y*; *sel-symm f*⟧ ⟹ *set-sel f X Y* = *set-sel f Y X*
⟨*proof*⟩

lemma *set-sel-aux-1-if-notfin*: ¬*finite Y* ⟹ *set-sel-aux f x Y* = *1*
⟨*proof*⟩

lemma *set-sel-1-if-notfin1*: ¬*finite X* ⟹ *set-sel f X Y* = *1*
⟨*proof*⟩

lemma *set-sel-1-if-notfin2*: $\neg$*finite* $Y \Longrightarrow$ *set-sel* $f\ X\ Y = 1$
⟨*proof*⟩

lemma *set-sel-symm*: *sel-symm* $f \Longrightarrow$ *set-sel* $f\ X\ Y =$ *set-sel* $f\ Y\ X$
⟨*proof*⟩

lemma *list-sel-aux'-eq-set-sel-aux'*:
distinct $xs \Longrightarrow$ *list-sel-aux'* $f\ xs\ x =$ *set-sel-aux'* f (*set* xs) x
⟨*proof*⟩

lemma *list-sel'-eq-set-sel'*:
⟦*distinct* xs; *distinct* ys⟧ $\Longrightarrow$ *list-sel'* $f\ xs\ ys =$ *set-sel'* f (*set* xs) (*set* ys)
⟨*proof*⟩

lemma *list-sel-eq-set-sel'*:
⟦*distinct* xs; *distinct* ys⟧ $\Longrightarrow$ *list-sel* $f\ xs\ ys =$ *set-sel'* f (*set* xs) (*set* ys)
⟨*proof*⟩

lemma *set-sel'-symm-if-finite*: ⟦*finite* X; *finite* Y; *sel-symm* f⟧ $\Longrightarrow$ *set-sel'* $f\ X\ Y$ $=$ *set-sel'* $f\ Y\ X$
⟨*proof*⟩

lemma *set-sel-aux'-1-if-notfin*: $\neg$*finite* $X \Longrightarrow$ *set-sel-aux'* $f\ X\ y = 1$
⟨*proof*⟩

lemma *set-sel'-1-if-notfin1*: $\neg$*finite* $X \Longrightarrow$ *set-sel'* $f\ X\ Y = 1$
⟨*proof*⟩

lemma *set-sel'-1-if-notfin2*: $\neg$*finite* $Y \Longrightarrow$ *set-sel'* $f\ X\ Y = 1$
⟨*proof*⟩

lemma *set-sel'-symm*: *sel-symm* $f \Longrightarrow$ *set-sel'* $f\ X\ Y =$ *set-sel'* $f\ Y\ X$
⟨*proof*⟩

lemma *set-sel'-eq-set-sel*: *set-sel'* $f\ X\ Y =$ *set-sel* $f\ X\ Y$
⟨*proof*⟩

lemma *set-sel-aux-reasonable-fin*:
⟦*finite* y; *sel-reasonable* f⟧ $\Longrightarrow$ *set-sel-aux* $f\ x\ y \leq 1 \wedge$ *set-sel-aux* $f\ x\ y > 0$
⟨*proof*⟩

lemma *set-sel-aux-reasonable*:
sel-reasonable $f \Longrightarrow$ *set-sel-aux* $f\ x\ y \leq 1 \wedge$ *set-sel-aux* $f\ x\ y > 0$
⟨*proof*⟩

lemma *set-sel-aux'-reasonable-fin*:
⟦*finite* x; *sel-reasonable* f⟧ $\Longrightarrow$ *set-sel-aux'* $f\ x\ y \leq 1 \wedge$ *set-sel-aux'* $f\ x\ y > 0$
⟨*proof*⟩

lemma *set-sel-aux′-reasonable*:
sel-reasonable f $\Longrightarrow$ *set-sel-aux′ f x y* $\leq$ *1* $\wedge$ *set-sel-aux′ f x y* $>$ *0*
⟨*proof*⟩

lemma *set-sel-reasonable-fin*:
⟦*finite x*; *sel-reasonable f*⟧ $\Longrightarrow$ *set-sel f x y* $\leq$ *1* $\wedge$ *set-sel f x y* $>$ *0*
⟨*proof*⟩

lemma *set-sel-reasonable*: *sel-reasonable f* $\Longrightarrow$ *set-sel f x y* $\leq$ *1* $\wedge$ *set-sel f x y* $>$ *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-sel′-reasonable*: *sel-reasonable f* $\Longrightarrow$ *set-sel′ f x y* $\leq$ *1* $\wedge$ *set-sel′ f x y* $>$ *0*
⟨*proof*⟩

lemma *ldeep-s-pos*: *sel-reasonable f* $\Longrightarrow$ *ldeep-s f xs x* $>$ *0*
⟨*proof*⟩

lemma *distinct-app-trans-r*: *distinct* (*ys*@*xs*) $\Longrightarrow$ *distinct xs*
⟨*proof*⟩

lemma *distinct-app-trans-l*: *distinct* (*ys*@*xs*) $\Longrightarrow$ *distinct ys*
⟨*proof*⟩

lemma *ldeep-s-reasonable*: *sel-reasonable f* $\Longrightarrow$ *ldeep-s f xs y* $\leq$ *1* $\wedge$ *ldeep-s f xs y* $>$ *0*
⟨*proof*⟩

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*⟩

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*⟩

lemma *distinct-ldeep-s-eq-aux′*:
⟦*distinct xs*; *as* @ *y* # *bs* = *xs*⟧ $\Longrightarrow$ *ldeep-s sel xs y* = *list-sel-aux′ sel bs y*
⟨*proof*⟩

lemma *ldeep-s-last1-if-distinct*: *distinct xs* $\Longrightarrow$ *ldeep-s sel xs* (*last xs*) = *1*
⟨*proof*⟩

lemma *ldeep-s-revhd1-if-distinct*: *distinct xs* $\Longrightarrow$ *ldeep-s sel* (*rev xs*) (*hd xs*) = *1*
⟨*proof*⟩

lemma *ldeep-s-1-if-nelem*: *x* $\notin$ *set xs* $\Longrightarrow$ *ldeep-s sel xs x* = *1*
⟨*proof*⟩

lemma *distinct-xs-not-ys*: *distinct* (*xs*@*ys*) $\Longrightarrow$ *x* $\in$ *set xs* $\Longrightarrow$ *x* $\notin$ *set ys*
⟨*proof*⟩

lemma *distinct-ys-not-xs*: *distinct* (*xs*@*ys*) $\Longrightarrow$ *x* $\in$ *set ys* $\Longrightarrow$ *x* $\notin$ *set xs*
⟨*proof*⟩

lemma *distinct-change-order-first-eq-nempty*:
assumes *distinct* (*xs*@*ys*@*zs*@*rs*)
and *ys* $\neq$ []
and *zs* $\neq$ []
and *take 1* (*xs*@*ys*@*zs*@*rs*) = *take 1* (*xs*@*zs*@*ys*@*rs*)
shows *xs* $\neq$ []
⟨*proof*⟩

lemma *distinct-change-order-first-elem*:
⟦*distinct* (*xs*@*ys*@*zs*@*rs*); *ys* $\neq$ []; *zs* $\neq$ []; *take 1* (*xs*@*ys*@*zs*@*rs*) = *take 1* (*xs*@*zs*@*ys*@*rs*)⟧
$\Longrightarrow$ *take 1* (*xs*@*ys*@*zs*@*rs*) = *take 1 xs*
⟨*proof*⟩

lemma *take1-singleton-app*: *take 1 xs* = [*r*] $\Longrightarrow$ *take 1* (*xs*@*ys*) = [*r*]
⟨*proof*⟩

lemma *hd-eq-take1*: *take 1 xs* = [*r*] $\Longrightarrow$ *hd xs* = *r*
⟨*proof*⟩

lemma *take1-eq-hd*: ⟦*xs* $\neq$ []; *hd xs* = *r*⟧ $\Longrightarrow$ *take 1 xs* = [*r*]
⟨*proof*⟩

lemma *nempty-if-take1*: *take 1 xs* = [*r*] $\Longrightarrow$ *xs* $\neq$ []
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *mirror-twice-id*[*simp*]: *mirror* (*mirror t*) = *t*
⟨*proof*⟩

lemma *rdeep-eq-ldeep*: *right-deep t* = *left-deep* (*mirror t*)
⟨*proof*⟩

lemma *mirror-zig-zag-preserv*: *zig-zag* (*mirror t*) = *zig-zag t*
⟨*proof*⟩

lemma *ldeep-zig-zag*: *left-deep t* ⟹ *zig-zag t*
⟨*proof*⟩

lemma *rdeep-zig-zag*: *right-deep t* ⟹ *zig-zag t*
⟨*proof*⟩

lemma *relations-nempty*: *relations t* ≠ {}
⟨*proof*⟩

lemma *set-implies-mset*: *x* ∈ *relations t* ⟹ *x* ∈# *relations-mset t*
⟨*proof*⟩

lemma *mset-implies-set*: *x* ∈# *relations-mset t* ⟹ *x* ∈ *relations t*
⟨*proof*⟩

lemma *inorder-eq-mset*: *mset* (*inorder t*) = *relations-mset t*
⟨*proof*⟩

lemma *relations-set-eq-mset*: *set-mset* (*relations-mset t*) = *relations t*
⟨*proof*⟩

lemma *inorder-eq-set*: *set* (*inorder t*) = *relations t*
⟨*proof*⟩

lemma *revorder-eq-mset*: *mset* (*revorder t*) = *relations-mset t*
⟨*proof*⟩

lemma *revorder-eq-set*: *set* (*revorder t*) = *relations t*
⟨*proof*⟩

lemma *revorder-eq-rev-inorder*: *revorder t* = *rev* (*inorder t*)
⟨*proof*⟩

lemma *inorder-eq-rev-revorder*: *inorder t* = *rev* (*revorder t*)
⟨*proof*⟩

lemma *mirror-mset-eq*[*simp*]: *relations-mset* (*mirror t*) = *relations-mset t*
⟨*proof*⟩

lemma *distinct-rels-alt*: *distinct-relations t* ⟷ *distinct* (*revorder t*)
⟨*proof*⟩

lemma *distinct-rels-alt′*:
distinct-relations t ⟷ (*let multi*=*relations-mset t in* ∀ *x*∈# *multi*. *count multi x* = *1*)
⟨*proof*⟩

lemma *inorder-nempty*: *inorder t* ≠ []
⟨*proof*⟩

lemma *revorder-nempty*: *revorder t* ≠ []
⟨*proof*⟩

lemma *mirror-distinct*: *distinct-relations t* ⟹ *distinct-relations* (*mirror t*)
⟨*proof*⟩

lemma *mirror-set-eq*[*simp*]: *relations* (*mirror t*) = *relations t*
⟨*proof*⟩

lemma *mirror-inorder-rev*: *inorder* (*mirror t*) = *rev* (*inorder t*)
⟨*proof*⟩

lemma *mirror-revorder-rev*: *revorder* (*mirror t*) = *rev* (*revorder t*)
⟨*proof*⟩

corollary *mirror-revorder-inorder*: *revorder* (*mirror t*) = *inorder t*

⟨*proof*⟩

corollary *mirror-inorder-revorder*: *inorder* (*mirror t*) = *revorder t*
⟨*proof*⟩

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

lemma *mirror-reasonable-cards*:
⟦*sel-symm f*; *reasonable-cards cf f t*⟧ ⟹ *reasonable-cards cf f* (*mirror t*)
⟨*proof*⟩

lemma *joinTree-cases*: (∃ *r*. *t*=(*Relation r*)) ∨ (∃ *l rr*. *t*=(*Join l* (*Relation rr*)))
∨ (∃ *l lr rr*. *t*=(*Join l* (*Join lr rr*)))
⟨*proof*⟩

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

lemma *ldeep-trans*: *left-deep* (*Join l r*) ⟹ *left-deep l*
⟨*proof*⟩

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*⟩

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*⟩

lemma *first-node-first-inorder*: ∃ *xs*. *inorder t* = *first-node t* # *xs*
⟨*proof*⟩

lemma *first-node-last-revorder*: ∃ *xs*. *revorder t* = *xs* @ [*first-node t*]
⟨*proof*⟩

lemma *first-node-eq-hd*: *first-node t* = *hd* (*inorder t*)
⟨*proof*⟩

lemma *distinct-elem-right-not-left*:
assumes *distinct-relations* (*Join l r*)
and *x* ∈ *relations r*

shows $x \notin$ *relations* l
⟨*proof*⟩

lemma *distinct-elem-left-not-right*:
assumes *distinct-relations* (*Join* l r)
and $x \in$ *relations* l
shows $x \notin$ *relations* r
⟨*proof*⟩

lemma *distinct-relations-disjoint*: *distinct-relations* (*Join* l r) $\Longrightarrow$ *relations* $l \cap$ *relations* $r = \{\}$
⟨*proof*⟩

lemma *distinct-trans-l*: *distinct-relations* (*Join* l r) $\Longrightarrow$ *distinct-relations* l
⟨*proof*⟩

lemma *distinct-trans-r*: *distinct-relations* (*Join* l r) $\Longrightarrow$ *distinct-relations* r
⟨*proof*⟩

lemma *distinct-and-disjoint-impl-count1*:
assumes *distinct-relations* l
and *distinct-relations* r
and *relations* $l \cap$ *relations* $r = \{\}$
and $x \in\#$ *relations-mset* (*Join* l r)
shows *count* (*relations-mset* (*Join* l r)) $x = 1$
⟨*proof*⟩

lemma *distinct-and-disjoint-impl-distinct*:
⟦*distinct-relations* l; *distinct-relations* r; *relations* $l \cap$ *relations* $r = \{\}$⟧
$\Longrightarrow$ *distinct-relations* (*Join* l r)
⟨*proof*⟩

lemma *reasonable-trans*:
reasonable-cards *cf* *f* (*Join* l r) $\Longrightarrow$ *reasonable-cards* *cf* *f* $l \wedge$ *reasonable-cards* *cf* *f* r
⟨*proof*⟩

lemma *mirror-height-eq*: *height* (*mirror* t) = *height* t
⟨*proof*⟩

lemma *height-0-rel*: *height* $t = 0 \Longrightarrow \exists\, r.\ t =$ *Relation* r
⟨*proof*⟩

lemma *height-gt-0-join*: *height* $t > 0 \Longrightarrow \exists\, l\ r.\ t =$ *Join* l r
⟨*proof*⟩

lemma *height-decr-l*: *height* (*Join* l r) > *height* l
⟨*proof*⟩

lemma *height-decr-r*: *height* (*Join l r*) > *height r*
⟨*proof*⟩

lemma *mirror-num-relations-eq*: *num-relations* (*mirror t*) = *num-relations t*
⟨*proof*⟩

lemma *zig-zag-num-relations-height*: *zig-zag t* ⟹ *num-relations t* = *height t* + *1*
⟨*proof*⟩

lemma *ldeep-num-relations-height*: *left-deep t* ⟹ *num-relations t* = *height t* + *1*
⟨*proof*⟩

lemma *rdeep-num-relations-height*: *right-deep t* ⟹ *num-relations t* = *height t* + *1*
⟨*proof*⟩

lemma *num-relations-eq-length*: *num-relations t* = *length* (*inorder t*)
⟨*proof*⟩

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

lemma *cards-list-eq-inorder*: *map* (λ(*a*,-). *a*) (*cards-list cf t*) = *inorder t*
⟨*proof*⟩

lemma *cards-list-eq-relations*: (λ(*a*,-). *a*) ‘ *set* (*cards-list cf t*) = *relations t*
⟨*proof*⟩

lemma *cards-eq-c*: (*rel*,*c*)∈*set*(*cards-list cf t*) ⟹ *cf rel* = *c*
⟨*proof*⟩

lemma *finite-trans*: *finite* (*relations* (*Join l r*)) ⟹ *finite* (*relations l*) ∧ *finite* (*relations r*)
⟨*proof*⟩

lemma *distinct-impl-card-eq-length*:
finite (*relations t*) ⟹ *height t* ≤ *n* ⟹ *distinct-relations t*
⟹ *Finite-Set.card* (*relations t*) = *length* (*inorder t*)
⟨*proof*⟩

lemma *card-le-length*: *Finite-Set.card* (*relations t*) ≤ *length* (*inorder t*)
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *card-eq-length-impl-distinct*:
⟦*finite* (*relations t*); *height t* ≤ *n*; *Finite-Set.card* (*relations t*) = *length* (*inorder t*)⟧
⟹ *distinct-relations t*
⟨*proof*⟩

lemma *list-sel-revorder-eq-inorder-x*: *list-sel f* (*revorder l*) *ys* = *list-sel f* (*inorder l*) *ys*
⟨*proof*⟩

lemma *list-sel-revorder-eq-inorder-y*: *list-sel f xs* (*revorder r*) = *list-sel f xs* (*inorder r*)
⟨*proof*⟩

lemma *list-sel-revorder-eq-inorder*:
list-sel f (*revorder l*) (*revorder r*) = *list-sel f* (*inorder l*) (*inorder r*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *distinct-alt*:
finite (*relations t*)
⟹ *distinct-relations t* ⟷ *Finite-Set.card* (*relations t*) = *length* (*inorder t*)
⟨*proof*⟩

lemma *distinct-alt2*:
distinct-relations (*Join l r*)
⟷ *distinct-relations l* ∧ *distinct-relations r* ∧ *relations l* ∩ *relations r* = {}
⟨*proof*⟩

lemma *pos-rel-cards-subtrees*:
pos-rel-cards cf (*Join l r*) = (*pos-rel-cards cf l* ∧ *pos-rel-cards cf r*)
⟨*proof*⟩

lemma *pos-rel-cards-eq-pos-list-cards*:

pos-rel-cards cf t $\longleftrightarrow$ *pos-list-cards cf* (*inorder t*)
⟨*proof*⟩

lemma *pos-list-cards-split*:
pos-list-cards cf (*xs*@*ys*) $\longleftrightarrow$ *pos-list-cards cf xs* $\wedge$ *pos-list-cards cf ys*
⟨*proof*⟩

lemma *pos-sel-reason-impl-reason*:
⟦*pos-rel-cards cf t*; *sel-reasonable sel*⟧ $\Longrightarrow$ *reasonable-cards cf sel t*
⟨*proof*⟩

lemma *create-rdeep-order*: *xs* $\neq$ [] $\Longrightarrow$ *inorder* (*create-rdeep xs*) = *xs*
⟨*proof*⟩

lemma *create-ldeep-rev-order*: *xs* $\neq$ [] $\Longrightarrow$ *inorder* (*create-ldeep-rev xs*) = *rev xs*
⟨*proof*⟩

lemma *create-ldeep-order*: *xs* $\neq$ [] $\Longrightarrow$ *inorder* (*create-ldeep xs*) = *xs*
⟨*proof*⟩

lemma *create-rdeep-rdeep*: *xs* $\neq$ [] $\Longrightarrow$ *right-deep* (*create-rdeep xs*)
⟨*proof*⟩

lemma *create-ldeep-rev-ldeep*: *xs* $\neq$ [] $\Longrightarrow$ *left-deep* (*create-ldeep-rev xs*)
⟨*proof*⟩

lemma *create-ldeep-ldeep*: *xs* $\neq$ [] $\Longrightarrow$ *left-deep* (*create-ldeep xs*)
⟨*proof*⟩

lemma *create-ldeep-rev-relations*: *xs* $\neq$ [] $\Longrightarrow$ *relations* (*create-ldeep-rev xs*) = *set xs*
⟨*proof*⟩

lemma *create-ldeep-relations*: *xs* $\neq$ [] $\Longrightarrow$ *relations* (*create-ldeep xs*) = *set xs*
⟨*proof*⟩

lemma *create-ldeep-rev-Cons*:
xs $\neq$ [] $\Longrightarrow$ *create-ldeep-rev* (*x*#*xs*) = *Join* (*create-ldeep-rev xs*) (*Relation x*)
⟨*proof*⟩

lemma *create-ldeep-snoc*: *xs* $\neq$ [] $\Longrightarrow$ *create-ldeep* (*xs*@[*x*]) = *Join* (*create-ldeep xs*) (*Relation x*)
⟨*proof*⟩

lemma *create-ldeep-inorder*[*simp*]: *left-deep t* $\Longrightarrow$ *create-ldeep* (*inorder t*) = *t*
⟨*proof*⟩

lemma *create-rdeep-inorder*[*simp*]: *right-deep t* $\Longrightarrow$ *create-rdeep* (*inorder t*) = *t*
⟨*proof*⟩

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*]
⟨*proof*⟩

lemma *ldeep-n-eq-card*:
⟦*distinct-relations t*; *left-deep t*⟧ ⟹ *ldeep-n f cf* (*revorder t*) = *card cf f t*
⟨*proof*⟩

lemma *ldeep-n-eq-card-subtree*:
⟦*distinct-relations* (*Join t r*′); *left-deep t*⟧ ⟹ *ldeep-n f cf* (*revorder t*) = *card cf f t*
⟨*proof*⟩

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*⟩

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

lemma *ldeep-T*′*-eq-foldl*: *acc* ∗ *ldeep-T*′ *f cf xs* = *foldl* (λ*a b*. *a* ∗ *cf b* ∗ *f b*) *acc xs*
⟨*proof*⟩

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*⟩

lemma *ldeep-T-eq-ldeep-T*′*-aux*: *ldeep-T sf cf xs* = *ldeep-T*′ *sf cf xs*
⟨*proof*⟩

lemma *ldeep-T-eq-ldeep-T*′: *ldeep-T* = *ldeep-T*′
⟨*proof*⟩

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

lemma *ldeep-T-app*: *ldeep-T f cf* (*xs*@*ys*) = *ldeep-T f cf xs* ∗ *ldeep-T f cf ys*
⟨*proof*⟩

lemma *ldeep-T-empty*: *ldeep-T f cf* [] = *1*

⟨*proof*⟩

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

lemma *ldeep-n-pos*: ⟦*pos-list-cards cf xs*; *sel-reasonable f*⟧ ⟹ *ldeep-n f cf xs* > *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ldeep-T-pos′*:
⟦*distinct xs*; *pos-list-cards cf xs*; *sel-reasonable f*⟧ ⟹ *ldeep-T* (*ldeep-s f xs*) *cf xs* > *0*
⟨*proof*⟩

lemma *ldeep-T-pos*: ⟦∀ *x*∈ *set ys*. *cf x* > *0*; *sel-reasonable f*⟧ ⟹ *ldeep-T* (*ldeep-s f xs*) *cf ys* > *0*
⟨*proof*⟩

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-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*) = (λ*a. 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*)
⟨*proof*⟩

lemma *c-nlj-symm*: *symmetric* (*c-nlj cf f*)
⟨*proof*⟩

lemma *c-smj-symm*: *symmetric* (*c-smj cf f*)
⟨*proof*⟩

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*⟩

theorem *c-hj-IKKBZ*: *left-deep t* ⟹ *c-hj cf f t* = *c-IKKBZ* (λ- -. *1.2*) *cf f t*
⟨*proof*⟩

lemma *change-fun-order*: *y*≠*rel*
⟹ (λ*a b*. *if a*=*rel then g a b else* (λ*c d*. *if c*=*y then h c d else f c d*) *a b*)
= (λ*a b*. *if a*=*y then h a b else* (λ*c d*. *if c*=*rel then g c d else f c d*) *a b*)
⟨*proof*⟩

lemma *c-IKKBZ-fun-notelem*:
assumes *left-deep t*
and *distinct-relations t*
and *y* ∉ *relations t*

and $f' = (\lambda a\ b.\ \mathit{if}\ a{=}y\ \mathit{then}\ z\ b\ \mathit{else}\ f\ a\ b)$
shows *c-IKKBZ f' cf sf t = c-IKKBZ f cf sf t*
⟨*proof*⟩

lemma *distinct-c-IKKBZ-ldeep-s-prepend*:
⟦*distinct*(*ys*@*revorder t*); *left-deep t*⟧
⟹ *c-IKKBZ* (λ*a b*. *ldeep-s f* (*ys*@*revorder t*) *a* ∗ *b*) *cf f t*
= *c-IKKBZ* (λ*a b*. *ldeep-s f* (*revorder t*) *a* ∗ *b*) *cf f t*
⟨*proof*⟩

lemma *distinct-c-IKKBZ-ldeep-s-subtree*:
assumes *distinct-relations* (*Join l* (*Relation rel*))
and *left-deep* (*Join l* (*Relation rel*))
shows *c-IKKBZ* (λ*a b*. *ldeep-s f* (*revorder* (*Join l* (*Relation rel*))) *a* ∗ *b*) *cf f l*
= *c-IKKBZ* (λ*a b*. *ldeep-s f* (*revorder l*) *a* ∗ *b*) *cf f l*
⟨*proof*⟩

theorem *c-out-IKKBZ*:
⟦*distinct-relations t*; *reasonable-cards cf f t*; *left-deep t*⟧
⟹ *c-IKKBZ* (λ*a b*. *ldeep-s f* (*revorder t*) *a* ∗ *b*) *cf f t* = *c-out cf f t*
⟨*proof*⟩

theorem *c-out-eq-c-list'*:
⟦*distinct-relations t*; *reasonable-cards cf f t*; *left-deep t*⟧
⟹ *c-list' f cf* (λ*xs x*. (*list-sel-aux' f xs x*) ∗ *cf x*) (*revorder t*) = *c-out cf f t*
⟨*proof*⟩

lemma *rev-first-last-elem*: (*rev* (*x*#*x'*#*xs'*)) = (*r*#*rs*) ⟹ *x* ∈# *mset rs*
⟨*proof*⟩

lemma *distinct-first-uneq-last*: *distinct* (*x*#*x'*#*xs'*) ⟹ *rev* (*x*#*x'*#*xs'*) = *r*#*rs*
⟹ *r* ≠ *x*
⟨*proof*⟩

lemma *distinct-create-eq-app*:
⟦*distinct* (*ys*@*xs*); *x* ∈# *mset xs*⟧ ⟹ *create-h-list h xs x* = *create-h-list h* (*ys*@*xs*)
x
⟨*proof*⟩

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*]
⟨*proof*⟩

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*]
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ldeep-s-h-eq-list-sel-aux′-h*:
⟦*distinct xs*; *ys*@*x*#*zs* = *xs*⟧
⟹ (λ*a*. *ldeep-s f xs a* ∗ *cf a*) *x* = (λ*xs x*. (*list-sel-aux′ f xs x*) ∗ *cf x*) *zs x*
⟨*proof*⟩

corollary *ldeep-s-h-eq-list-sel-aux′-h′*:
⟦*distinct-relations t*; *ys*@*x*#*zs* = *revorder t*⟧
⟹ (λ*a*. *ldeep-s f* (*revorder t*) *a* ∗ *cf a*) *x* = (λ*xs x*. (*list-sel-aux′ f xs x*) ∗ *cf x*) *zs x*
⟨*proof*⟩

lemma *create-h-list-distinct-simp*: ⟦*distinct xs*; *ys*@*x*#*zs* = *xs*⟧ ⟹ *create-h-list h xs x* = *h zs x*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

corollary *ldeep-s-h-eq-create-h-list''*:
distinct-relations t $\Longrightarrow \forall$ *ys x zs. ys@x#zs = revorder t*
$\longrightarrow$ *(λa. ldeep-s f (revorder t) a * cf a) x*
*= create-h-list (λxs x. (list-sel-aux' f xs x) * cf x) (revorder t) x*
⟨*proof*⟩

lemma *ldeep-s-h-eq-create-h-list'''*:
⟦*distinct-relations t*; *x* ∈ *relations t*⟧
$\Longrightarrow$ *(λa. ldeep-s f (revorder t) a * cf a) x*
*= create-h-list (λxs x. (list-sel-aux' f xs x) * cf x) (revorder t) x*
⟨*proof*⟩

lemma *cons2-if-2elems*: ⟦*x* ∈ *set xs*; *y* ∈ *set xs*; *x* ≠ *y*⟧ $\Longrightarrow \exists$ *y z zs. xs = y # z # zs*
⟨*proof*⟩

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 $\forall$ *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*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *create-h-list-pos*:
⟦*sel-reasonable sf*; ∀ *x* ∈ *set xs*. *cf x* > *0*⟧
⟹ (*create-h-list* (λ*xs x*. (*list-sel-aux′ sf xs x*) ∗ *cf x*) *xs*) *x* > *0*
⟨*proof*⟩

lemma *c-list-not-neg*:
assumes *sel-reasonable sf*
and ∀ *x* ∈ *set ys*. *cf x* > *0*
and *h* = (λ*a*. *ldeep-s sf xs a* ∗ *cf a*)
shows *c-list* (*ldeep-s sf xs*) *cf h r ys* ≥ *0*
⟨*proof*⟩

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* (λ*xs x*. (*list-sel-aux′ sf xs x*) ∗ *cf x*) *xs*
shows *c-list* (*ldeep-s sf xs*) *cf h r ys* ≥ *0*
⟨*proof*⟩

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*⟩

lemma *c-list-pos-r-not-elem*:
assumes *sel-reasonable sf*
and $\forall x \in set\ ys.\ cf\ x > 0$
and $ys \neq []$
and $r \notin set\ ys$
and $h = (\lambda a.\ ldeep\text{-}s\ sf\ xs\ a * cf\ a)$
shows *c-list* $(ldeep\text{-}s\ sf\ xs)\ cf\ h\ r\ ys > 0$
⟨*proof*⟩

lemma *c-list-pos-r-not-elem-hlist*:
assumes *sel-reasonable sf*
and $\forall x \in set\ xs.\ cf\ x > 0$
and $\forall x \in set\ ys.\ cf\ x > 0$
and $ys \neq []$
and $r \notin set\ ys$
and $h = create\text{-}h\text{-}list\ (\lambda xs\ x.\ (list\text{-}sel\text{-}aux'\ sf\ xs\ x) * cf\ x)\ xs$
shows *c-list* $(ldeep\text{-}s\ sf\ xs)\ cf\ h\ r\ ys > 0$
⟨*proof*⟩

lemma *c-list-pos-not-root*:
assumes *sel-reasonable sf*
and $\forall x \in set\ ys.\ cf\ x > 0$
and $ys \neq []$
and $ys \neq [r]$
and *distinct ys*
and $h = (\lambda a.\ ldeep\text{-}s\ sf\ xs\ a * cf\ a)$
shows *c-list* $(ldeep\text{-}s\ sf\ xs)\ cf\ h\ r\ ys > 0$
⟨*proof*⟩

lemma *c-list-pos-not-root-hlist*:
assumes *sel-reasonable sf*
and $\forall x \in set\ xs.\ cf\ x > 0$
and $\forall x \in set\ ys.\ cf\ x > 0$
and $ys \neq []$
and $ys \neq [r]$
and *distinct ys*
and $h = create\text{-}h\text{-}list\ (\lambda xs\ x.\ (list\text{-}sel\text{-}aux'\ sf\ xs\ x) * cf\ x)\ xs$
shows *c-list* $(ldeep\text{-}s\ sf\ xs)\ cf\ h\ r\ ys > 0$
⟨*proof*⟩

lemma *c-list-split-four*:
assumes $T = ldeep\text{-}T\ f\ cf$
and $C = c\text{-}list\ f\ cf\ h\ r$
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*⟩

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*⟩

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*⟩

lemma *c-list-pos-asi*:
fixes *sf cf h r xs*
defines *f ≡ ldeep-s sf xs*
defines *rank ≡ (λl. (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)*
⟨*proof*⟩

theorem *c-list-asi*:
fixes *sf cf h r xs*
defines *f ≡ ldeep-s sf xs*
defines *rank ≡ (λl. (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)*
⟨*proof*⟩

corollary *c-out-asi*:
fixes *sf cf r xs*

defines $f \equiv$ *ldeep-s sf xs*
defines $h \equiv (\lambda a.$ *ldeep-s sf xs* $a *$ *cf* $a)$
defines *rank* $\equiv (\lambda l.$ *(ldeep-T f cf* $l - 1)$ */ c-list f cf h r l)*
assumes *sel-reasonable sf*
and $\forall x.$ *cf* $x > 0$
shows *asi rank r (c-list f cf h r)*
⟨*proof*⟩

lemma *c-out-asi-aux*:
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 $h = (\lambda a.$ *ldeep-s sf xs* $a *$ *cf* $a)$
and *distinct (A@U@V@B)*
and $U \neq []$
and $V \neq []$
and *rank* $= (\lambda l.$ *(ldeep-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*⟩

lemma *c-out-asi-aux-hlist*:
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* $(\lambda xs\ x.$ *(list-sel-aux′ sf xs x)* $*$ *cf* $x)$ *xs*
and *distinct (A@U@V@B)*
and $U \neq []$
and $V \neq []$
and *rank* $= (\lambda l.$ *(ldeep-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*⟩

theorem *c-out-asi-altproof*:
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 $h = (\lambda a.$ *ldeep-s sf xs* $a *$ *cf* $a)$
shows *asi* $(\lambda l.$ *(ldeep-T f cf* $l - 1)$ */ c l) r (c-list f cf h r)*
⟨*proof*⟩

theorem *c-out-asi-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 $h = create\text{-}h\text{-}list\ (\lambda xs\ x.\ (list\text{-}sel\text{-}aux'\ sf\ xs\ x) * cf\ x)\ xs$
shows $asi\ (\lambda l.\ (ldeep\text{-}T\ f\ cf\ l - 1)\ /\ c\ l)\ r\ (c\text{-}list\ f\ cf\ h\ r)$
⟨*proof*⟩

lemma *asi-if-asi'*: $asi\ rank\ r\ c \Longrightarrow asi'\ r\ c$
⟨*proof*⟩

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 $asi'\ r\ (c\text{-}list\ f\ cf\ h\ r)$
⟨*proof*⟩

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 $asi'\ r\ (c\text{-}list\ f\ cf\ h\ r)$
⟨*proof*⟩

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*⟩

theorem *c-out-asi''*:
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 $h = create\text{-}h\text{-}list\ (\lambda xs\ x.\ (list\text{-}sel\text{-}aux'\ sf\ xs\ x) * cf\ x)\ xs$
shows $asi''\ (\lambda l.\ (ldeep\text{-}T\ f\ cf\ l - 1)\ /\ c\ l)\ r\ (c\text{-}list\ f\ cf\ h\ r)$
⟨*proof*⟩

3.4.2 Additional ASI Proofs

lemma *asi-le-iff-notr*:
$[\![asi\ rank\ r\ cost;\ U \neq [];\ V \neq [];\ r \notin set\ (A\ @\ U\ @\ V\ @\ B);\ distinct\ (A\ @\ U\ @\ V\ @\ B)]\!]$
$\Longrightarrow rank\ (rev\ U) \leq rank\ (rev\ V) \longleftrightarrow cost\ (rev\ (A@U@V@B)) \leq cost\ (rev\ (A@V@U@B))$
⟨*proof*⟩

lemma *asi-le-iff-rfst*:
$[\![asi\ rank\ r\ cost;\ U \neq [];\ V \neq [];$
$take\ 1\ (A\ @\ U\ @\ V\ @\ B) = [r];\ take\ 1\ (A\ @\ V\ @\ U\ @\ B) = [r];\ distinct\ (A\ @\ U\ @\ V\ @\ B)]\!]$
$\Longrightarrow rank\ (rev\ U) \leq rank\ (rev\ V) \longleftrightarrow cost\ (rev\ (A@U@V@B)) \leq cost\ (rev\ (A@V@U@B))$
⟨*proof*⟩

lemma *asi-le-notr*:
$[\![asi\ rank\ r\ cost;\ rank\ (rev\ U) \leq rank\ (rev\ V);\ U{\neq}[];\ V{\neq}[];$
$distinct\ (A@U@V@B);\ r \notin set\ (A@U@V@B)]\!]$
$\Longrightarrow cost\ (rev\ (A@U@V@B)) \leq cost\ (rev\ (A@V@U@B))$
⟨*proof*⟩

lemma *asi-le-rfst*:
$[\![asi\ rank\ r\ cost;\ rank\ (rev\ U) \leq rank\ (rev\ V);\ U{\neq}[];\ V{\neq}[];\ distinct\ (A@U@V@B);$
$take\ 1\ (A\ @\ U\ @\ V\ @\ B) = [r];\ take\ 1\ (A\ @\ V\ @\ U\ @\ B) = [r]]\!]$
$\Longrightarrow cost\ (rev\ (A@U@V@B)) \leq cost\ (rev\ (A@V@U@B))$
⟨*proof*⟩

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*⟩

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*⟩

lemma *asi-eq-iff-notr*:
⟦*asi rank r cost*; $U \neq []$; $V \neq []$; $r \notin set\ (A@U@V@B)$; *distinct* $(A@U@V@B)$⟧
$\Longrightarrow rank\ (rev\ U) = rank\ (rev\ V) \longleftrightarrow cost\ (rev\ (A@U@V@B)) = cost\ (rev\ (A@V@U@B))$
⟨*proof*⟩

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*⟩

lemma *asi-eq-rfst'*:
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 $take\ 1\ (A\ @\ U\ @\ V\ @\ B) = [r]$
and $take\ 1\ (A\ @\ V\ @\ U\ @\ B) = [r]$
and *distinct* $(A\ @\ U\ @\ V\ @\ B)$
shows $rank\ (rev\ U) = rank\ (rev\ V)$
⟨*proof*⟩

lemma *asi-eq-iff-rfst*:
⟦*asi rank r cost*; $U \neq []$; $V \neq []$;
$take\ 1\ (A\ @\ U\ @\ V\ @\ B) = [r]$; $take\ 1\ (A\ @\ V\ @\ U\ @\ B) = [r]$; *distinct* $(A\ @\ U\ @\ V\ @\ B)$⟧
$\Longrightarrow rank\ (rev\ U) = rank\ (rev\ V) \longleftrightarrow cost\ (rev\ (A@U@V@B)) = cost\ (rev\ (A@V@U@B))$
⟨*proof*⟩

lemma *asi-lt-iff-notr*:
assumes *asi rank r cost*
and $U \neq []$ **and** $V \neq []$
and $r \notin set\ (A\ @\ U\ @\ V\ @\ B)$
and *distinct* $(A\ @\ U\ @\ V\ @\ B)$
shows $rank\ (rev\ U) < rank\ (rev\ V) \longleftrightarrow cost\ (rev\ (A@U@V@B)) < cost\ (rev\ (A@V@U@B))$
⟨*proof*⟩

lemma *asi-lt-iff-rfst*:
assumes *asi rank r cost*
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 *rank* $(rev\ U) < rank\ (rev\ V) \longleftrightarrow cost\ (rev\ (A@U@V@B)) < cost\ (rev\ (A@V@U@B))$
⟨*proof*⟩

lemma *asi-lt-notr*:
$[\![asi\ rank\ r\ cost;\ rank\ (rev\ U) < rank\ (rev\ V);\ U{\neq}[];\ V{\neq}[];$
$distinct\ (A@U@V@B);\ r \notin set\ (A@U@V@B)]\!]$
$\Longrightarrow cost\ (rev\ (A@U@V@B)) < cost\ (rev\ (A@V@U@B))$
⟨*proof*⟩

lemma *asi-lt-rfst*:
$[\![asi\ rank\ r\ cost;\ rank\ (rev\ U) < rank\ (rev\ V);\ U{\neq}[];\ V{\neq}[];\ distinct\ (A@U@V@B);$
$take\ 1\ (A \ @ \ U \ @ \ V \ @ \ B) = [r];\ take\ 1\ (A \ @ \ V \ @ \ U \ @ \ B) = [r]]\!]$
$\Longrightarrow cost\ (rev\ (A@U@V@B)) < cost\ (rev\ (A@V@U@B))$
⟨*proof*⟩

lemma *asi''-simp-iff*:
$[\![asi''\ rank\ r\ cost;\ U \neq [];\ V \neq [];\ U \neq [r];\ V \neq [r];\ distinct\ (A \ @ \ U \ @ \ V \ @ \ B)]\!]$
$\Longrightarrow rank\ (rev\ U) \leq rank\ (rev\ V) \longleftrightarrow cost\ (rev\ (A@U@V@B)) \leq cost\ (rev\ (A@V@U@B))$
⟨*proof*⟩

lemma *asi''-simp*:
$[\![asi''\ rank\ r\ cost;\ rank\ (rev\ U) \leq rank\ (rev\ V);\ U{\neq}[];\ V{\neq}[];\ distinct\ (A@U@V@B);$
$U{\neq}[r];\ V{\neq}[r]]\!]$
$\Longrightarrow cost\ (rev\ (A@U@V@B)) \leq cost\ (rev\ (A@V@U@B))$
⟨*proof*⟩

end

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

lemma *two-elems-card-ge-2*: *finite xs* $\Longrightarrow x \in xs \wedge y \in xs \wedge x{\neq}y \Longrightarrow$ *Finite-Set.card xs* ≥ 2
⟨*proof*⟩

4 Graph Extensions

context *wf-digraph*

begin

lemma *awalk-dom-if-uneq*: $[\![u{\neq}v;\ awalk\ u\ p\ v]\!] \Longrightarrow \exists\, x.\ x \rightarrow_G v$
⟨*proof*⟩

lemma *awalk-verts-dom-if-uneq*: $[\![u{\neq}v;\ awalk\ u\ p\ v]\!] \Longrightarrow \exists\, x.\ x \rightarrow_G v \wedge x \in set\ (awalk\text{-}verts\ u\ p)$
⟨*proof*⟩

lemma *awalk-verts-append-distinct*:
$[\![\exists\, v.\ awalk\ r\ (p1@p2)\ v;\ distinct\ (awalk\text{-}verts\ r\ (p1@p2))]\!] \Longrightarrow distinct\ (awalk\text{-}verts\ r\ p1)$
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *awalk-verts-subset-if-p-sub*:
$[\![awalk\ u\ p1\ v;\ awalk\ u\ p2\ v;\ set\ p1 \subseteq set\ p2]\!]$
$\Longrightarrow set\ (awalk\text{-}verts\ u\ p1) \subseteq set\ (awalk\text{-}verts\ u\ p2)$
⟨*proof*⟩

lemma *awalk-to-apath-verts-subset*:
$awalk\ u\ p\ v \Longrightarrow set\ (awalk\text{-}verts\ u\ (awalk\text{-}to\text{-}apath\ p)) \subseteq set\ (awalk\text{-}verts\ u\ p)$
⟨*proof*⟩

lemma *unique-apath-verts-in-awalk*:
$[\![x \in set\ (awalk\text{-}verts\ u\ p1);\ apath\ u\ p1\ v;\ awalk\ u\ p2\ v;\ \exists!p.\ apath\ u\ p\ v]\!]$
$\Longrightarrow x \in set\ (awalk\text{-}verts\ u\ p2)$
⟨*proof*⟩

lemma *unique-apath-verts-sub-awalk*:
$[\![apath\ u\ p\ v;\ awalk\ u\ q\ v;\ \exists!p.\ apath\ u\ p\ v]\!] \Longrightarrow set\ (awalk\text{-}verts\ u\ p) \subseteq set\ (awalk\text{-}verts\ u\ q)$
⟨*proof*⟩

lemma *awalk-verts-append3*:
$[\![awalk\ u\ (p@e\#q)\ r;\ awalk\ v\ q\ r]\!] \Longrightarrow awalk\text{-}verts\ u\ (p@e\#q) = awalk\text{-}verts\ u\ p\ @\ awalk\text{-}verts\ v\ q$
⟨*proof*⟩

lemma *verts-reachable-connected*:
$verts\ G \neq \{\} \Longrightarrow (\forall\, x{\in}verts\ G.\ \forall\, y{\in}verts\ G.\ x \rightarrow^* y) \Longrightarrow connected\ G$
⟨*proof*⟩

lemma *out-degree-0-no-arcs*:
assumes *out-degree G v* = *0* **and** *finite* (*arcs G*)

shows $\forall y.\ (v,y) \notin \textit{arcs-ends}\ G$
⟨*proof*⟩

lemma *out-degree-0-only-self*: *finite* (*arcs* G) $\Longrightarrow$ *out-degree* $G\ v = 0 \Longrightarrow v \rightarrow^* x \Longrightarrow x = v$
⟨*proof*⟩

lemma *not-elem-no-out-arcs*: $v \notin \textit{verts}\ G \Longrightarrow \textit{out-arcs}\ G\ v = \{\}$
⟨*proof*⟩

lemma *not-elem-no-in-arcs*: $v \notin \textit{verts}\ G \Longrightarrow \textit{in-arcs}\ G\ v = \{\}$
⟨*proof*⟩

lemma *not-elem-out-0*: $v \notin \textit{verts}\ G \Longrightarrow \textit{out-degree}\ G\ v = 0$
⟨*proof*⟩

lemma *not-elem-in-0*: $v \notin \textit{verts}\ G \Longrightarrow \textit{in-degree}\ G\ v = 0$
⟨*proof*⟩

lemma *new-vert-only-no-arcs*:
assumes $G = (\!|\textit{verts} = V \cup \{v\},\ \textit{arcs} = A,\ \textit{tail} = t,\ \textit{head} = h|\!)$
and $G' = (\!|\textit{verts} = V,\ \textit{arcs} = A,\ \textit{tail} = t,\ \textit{head} = h|\!)$
and *wf-digraph* G'
and $v \notin V$
and *finite* (*arcs* G)
shows $\forall u.\ (v,u) \notin \textit{arcs-ends}\ G$
⟨*proof*⟩

lemma *new-leaf-out-sets-eq*:
assumes $G = (\!|\textit{verts} = V \cup \{v\},\ \textit{arcs} = A \cup \{a\},\ \textit{tail} = t(a := u),\ \textit{head} = h(a := v)|\!)$
and $G' = (\!|\textit{verts} = V,\ \textit{arcs} = A,\ \textit{tail} = t,\ \textit{head} = h|\!)$
and $u \in V$
and $v \notin V$
and $a \notin A$
shows $\{e \in \textit{arcs}\ G.\ \textit{tail}\ G\ e = v\} = \{e \in \textit{arcs}\ G'.\ \textit{tail}\ G'\ e = v\}$
⟨*proof*⟩

lemma *new-leaf-out-0*:
assumes $G = (\!|\textit{verts} = V \cup \{v\},\ \textit{arcs} = A \cup \{a\},\ \textit{tail} = t(a := u),\ \textit{head} = h(a := v)|\!)$
and $G' = (\!|\textit{verts} = V,\ \textit{arcs} = A,\ \textit{tail} = t,\ \textit{head} = h|\!)$
and *wf-digraph* G'
and $u \in V$
and $v \notin V$
and $a \notin A$
shows *out-degree* $G\ v = 0$
⟨*proof*⟩

lemma *new-leaf-no-arcs*:
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 $a \notin A$
and *finite* (*arcs* G)
shows $\forall u.\ (v,u) \notin$ *arcs-ends* G
⟨*proof*⟩

lemma *tail-and-head-eq-impl-cas*:
assumes *cas* $x\ p\ y$
and $\forall x \in$ *set* $p.$ *tail* $G\ x =$ *tail* $G'\ x$
and $\forall x \in$ *set* $p.$ *head* $G\ x =$ *head* $G'\ x$
shows *pre-digraph.cas* $G'\ x\ p\ y$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*))
⟨*proof*⟩

lemma *graph-del-vert*: *graph* (*del-vert x*)
⟨*proof*⟩

lemma *connected-iff-reachable*:
connected G ⟷ ((∀ *x*∈*verts G*. ∀ *y*∈*verts G*. *x* →* *y*) ∧ *verts G* ≠ {})
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *last-branch-is-branch*:
(*y*∈*last-branching-points* ⟹ *y*∈*branching-points*)
⟨*proof*⟩

lemma *last-branch-alt*: *x* ∈ *last-branching-points* ⟹ (∀ *z*. *x* →* *z* ∧ *z*≠*x* ⟶ *z* ∉ *branching-points*)
⟨*proof*⟩

lemma *braching-points-alt*:
assumes *finite* (*arcs G*)
shows *x* ∈ *branching-points* ⟷ *out-degree G x* ≥ *2* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *branch-in-supergraph*:
assumes *subgraph C G*
and *x* ∈ *wf-digraph.branching-points C*
shows *x* ∈ *branching-points*
⟨*proof*⟩

lemma *subgraph-no-branch-chain*:
assumes *subgraph C G*
and *verts C* ⊆ *verts G* − {*x*. ∃ *y*∈*branching-points*. *x* →*$_G$* *y*}
shows *wf-digraph.is-chain C*
⟨*proof*⟩

lemma *branch-if-leaf-added*:
assumes *x*∈*wf-digraph.branching-points G'*
and *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 *a* ∉ *A*
shows *x* ∈ *branching-points*
⟨*proof*⟩

lemma *new-leaf-no-branch*:
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 $u \in V$
and $v \notin V$
and $a \notin A$
shows $v \notin$ *branching-points*
⟨*proof*⟩

lemma *new-leaf-not-reach-last-branch*:
assumes $y \in$*wf-digraph.last-branching-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-points*. $z \neq y \wedge y \rightarrow^* z)$
⟨*proof*⟩

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*⟩

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*⟩

end

4.2 Vertices with Multiple Incoming Arcs

context *wf-digraph*
begin

definition *merging-points* :: *'a set* **where**
merging-points = {*x*. ∃ *y*∈*arcs G*. ∃ *z*∈*arcs G*. *y*≠*z* ∧ *head G y* = *x* ∧ *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*∈*merging-points* ∧ ¬(∃ *y* ∈ *merging-points*. *y*≠*x* ∧ *x* →* *y*))}

lemma *merge-in-verts*: *x* ∈ *merging-points* ⟹ *x* ∈ *verts G*
⟨*proof*⟩

lemma *last-merge-is-merge*:
(*y*∈*last-merging-points* ⟹ *y*∈*merging-points*)
⟨*proof*⟩

lemma *last-merge-alt*: *x* ∈ *last-merging-points* ⟹ (∀ *z*. *x* →* *z* ∧ *z*≠*x* ⟶ *z* ∉ *merging-points*)
⟨*proof*⟩

lemma *merge-in-supergraph*:
assumes *subgraph C G*
and *x* ∈ *wf-digraph.merging-points C*
shows *x* ∈ *merging-points*
⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

lemma *joinTree-card-pos*: *matching-rels t* ⟹ *pos-rel-cards cf t*

⟨*proof*⟩

lemma *symmetric-arcs*: $x \in arcs\ G \Longrightarrow \exists\, y.\ head\ G\ x = tail\ G\ y \wedge tail\ G\ x = head\ G\ y$
⟨*proof*⟩

lemma *arc-ends-eq-impl-sel-eq*: $head\ G\ x = head\ G\ y \Longrightarrow tail\ G\ x = tail\ G\ y \Longrightarrow sel\ x = sel\ y$
⟨*proof*⟩

lemma *arc-ends-eq-impl-arc-eq*:
$[\![e1 \in arcs\ G;\ e2 \in arcs\ G;\ head\ G\ e1 = head\ G\ e2;\ tail\ G\ e1 = tail\ G\ e2]\!] \Longrightarrow e1 = e2$
⟨*proof*⟩

lemma *matching-sel-simp-if-not1*:
$[\![matching\text{-}sel\ sf;\ sf\ x\ y \neq 1]\!] \Longrightarrow \exists\, e \in arcs\ G.\ tail\ G\ e = x \wedge head\ G\ e = y \wedge sf\ x\ y = sel\ e$
⟨*proof*⟩

lemma *matching-sel-simp-if-arc*:
$[\![matching\text{-}sel\ sf;\ e \in arcs\ G]\!] \Longrightarrow sf\ (tail\ G\ e)\ (head\ G\ e) = sel\ e$
⟨*proof*⟩

lemma *matching-sel1-if-no-arc*: $matching\text{-}sel\ sf \Longrightarrow \neg(x \rightarrow_G y \vee y \rightarrow_G x) \Longrightarrow sf\ x\ y = 1$
⟨*proof*⟩

lemma *matching-sel-alt-aux1*:
$matching\text{-}sel\ f$
$\Longrightarrow (\forall\, x\ y.\ (\exists\, e \in arcs\ G.\ (tail\ G\ e) = x \wedge (head\ G\ e) = y \wedge f\ x\ y = sel\ e)$
$\vee\ ((\nexists\, e.\ e \in arcs\ G \wedge (tail\ G\ e) = x \wedge (head\ G\ e) = y) \wedge f\ x\ y = 1))$
⟨*proof*⟩

lemma *matching-sel-alt-aux2*:
$(\forall\, x\ y.(\exists\, e \in arcs\ G.\ (tail\ G\ e) = x \wedge (head\ G\ e) = y \wedge f\ x\ y = sel\ e)$
$\vee\ ((\nexists\, e.\ e \in arcs\ G \wedge (tail\ G\ e) = x \wedge (head\ G\ e) = y) \wedge f\ x\ y = 1))$
$\Longrightarrow matching\text{-}sel\ f$
⟨*proof*⟩

lemma *matching-sel-alt*:
$matching\text{-}sel\ f$
$= (\forall\, x\ y.\ (\exists\, e \in arcs\ G.\ (tail\ G\ e) = x \wedge (head\ G\ e) = y \wedge f\ x\ y = sel\ e)$
$\vee\ ((\nexists\, e.\ e \in arcs\ G \wedge (tail\ G\ e) = x \wedge (head\ G\ e) = y) \wedge f\ x\ y = 1))$
⟨*proof*⟩

lemma *matching-sel-symm*:
assumes *matching-sel f*
shows *sel-symm f*

⟨*proof*⟩

lemma *matching-sel-reasonable*: *matching-sel f* ⟹ *sel-reasonable f*
⟨*proof*⟩

lemma *matching-reasonable-cards*:
⟦*matching-sel f*; *matching-rels t*⟧ ⟹ *reasonable-cards cf f t*
⟨*proof*⟩

lemma *matching-sel-unique-aux*:
assumes *matching-sel f matching-sel g*
shows *f x y* = *g x y*
⟨*proof*⟩

lemma *matching-sel-unique*: ⟦*matching-sel f*; *matching-sel g*⟧ ⟹ *f* = *g*
⟨*proof*⟩

lemma *match-sel-matching*[*intro*]: *matching-sel match-sel*
⟨*proof*⟩

corollary *match-sel-unique*: *matching-sel f* ⟹ *f* = *match-sel*
⟨*proof*⟩

corollary *match-sel1-if-no-arc*: ¬(*x* →$_G$ *y* ∨ *y* →$_G$ *x*) ⟹ *match-sel x y* = *1*
⟨*proof*⟩

corollary *match-sel-symm*[*intro*]: *sel-symm match-sel*
⟨*proof*⟩

corollary *match-sel-reasonable*[*intro*]: *sel-reasonable match-sel*
⟨*proof*⟩

corollary *match-reasonable-cards*: *matching-rels t* ⟹ *reasonable-cards cf match-sel t*
⟨*proof*⟩

lemma *matching-rels-trans*: *matching-rels* (*Join l r*) = (*matching-rels l* ∧ *matching-rels r*)
⟨*proof*⟩

lemma *first-node-in-verts-if-rels-eq-verts*: *relations t* = *verts G* ⟹ *first-node t* ∈ *verts G*
⟨*proof*⟩

lemma *first-node-in-verts-if-valid*: *valid-tree t* ⟹ *first-node t* ∈ *verts G*
⟨*proof*⟩

lemma *dominates-sym*: (*x* →$_G$ *y*) ⟷ (*y* →$_G$ *x*)
⟨*proof*⟩

lemma *no-cross-mirror-eq*: *no-cross-products* (*mirror t*) = *no-cross-products t*
⟨*proof*⟩

lemma *no-cross-create-ldeep-rev-app*:
⟦*ys*≠[]; *no-cross-products* (*create-ldeep-rev* (*xs*@*ys*))⟧ ⟹ *no-cross-products* (*create-ldeep-rev ys*)
⟨*proof*⟩

lemma *no-cross-create-ldeep-app*:
⟦*xs*≠[]; *no-cross-products* (*create-ldeep* (*xs*@*ys*))⟧ ⟹ *no-cross-products* (*create-ldeep xs*)
⟨*proof*⟩

lemma *matching-rels-if-no-cross*: ⟦∀ *r*. *t* ≠ *Relation r*; *no-cross-products t*⟧ ⟹ *matching-rels t*
⟨*proof*⟩

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*⟩

lemma *no-cross-apath*:
⟦*matching-rels t*; *no-cross-products t*; *x* ∈ *relations t*; *y* ∈ *relations t*⟧
⟹ ∃ *p*. *apath x p y* ∧ *set* (*awalk-verts x p*) ⊆ *relations t*
⟨*proof*⟩

lemma *no-cross-reachable*:
⟦*matching-rels t*; *no-cross-products t*; *x* ∈ *relations t*; *y* ∈ *relations t*⟧ ⟹ *x* →* *y*
⟨*proof*⟩

corollary *reachable-if-no-cross*:
⟦∃ *t*. *relations t* = *verts G* ∧ *no-cross-products t*; *x* ∈ *verts G*; *y* ∈ *verts G*⟧ ⟹ *x* →* *y*
⟨*proof*⟩

lemma *remove-sel-sym*:
⟦*tail G* e_1 = *head G* e_2; *head G* e_1 = *tail G* e_2⟧ ⟹ (*remove-sel x*) e_1 = (*remove-sel x*) e_2
⟨*proof*⟩

lemma *remove-sel-1*: *e* ∉ *arcs G* ⟹ (*remove-sel x*) *e* = *1*
⟨*proof*⟩

lemma *del-vert-remove-sel-1*:
assumes *e* ∉ *arcs* ((*del-vert x*))
shows (*remove-sel x*) *e* = *1*
⟨*proof*⟩

lemma *remove-sel-pos*: *remove-sel x e* > *0*
⟨*proof*⟩

lemma *remove-sel-leq-1*: *remove-sel x e* ≤ *1*
⟨*proof*⟩

lemma *del-vert-pos-cards*: *x* ∈ *verts* (*del-vert y*) ⟹ *cf x* > *0*
⟨*proof*⟩

lemma *del-vert-remove-sel-query-graph*:
query-graph G sel cf ⟹ *query-graph* (*del-vert x*) (*remove-sel x*) *cf*
⟨*proof*⟩

lemma *finite-nempty-set-min*:
assumes *xs* ≠ {} **and** *finite xs*
shows ∃ *x*. *min-degree xs x*
⟨*proof*⟩

lemma *no-cross-reachable-graph'*:
⟦∃ *t*. *relations t* = *verts G* ∧ *no-cross-products t*; *x*∈*verts G*; *y*∈*verts G*⟧
⟹ *x* →**mk-symmetric G y*
⟨*proof*⟩

lemma *verts-nempty-if-tree*: ∃ *t*. *relations t* ⊆ *verts G* ⟹ *verts G* ≠ {}
⟨*proof*⟩

lemma *connected-if-tree*: ∃ *t*. *relations t* = *verts G* ∧ *no-cross-products t* ⟹ *connected G*
⟨*proof*⟩

end

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

5.3 Pair Query Graph

Alternative definition based on pair graphs

locale *pair-query-graph* = *pair-graph* +
fixes *sel* :: ('*a* × '*a*) *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* ∉ *parcs G* ⟹ *sel e* = *1*
and *sel-pos*: *sel e* > *0*
and *sel-leq-1*: *sel e* ≤ *1*
and *pos-cards*: *x* ∈ *pverts G* ⟹ *cf x* > *0*

sublocale *pair-query-graph* $\subseteq$ *query-graph*
⟨*proof*⟩

context *pair-query-graph*
begin

lemma *matching-sel f* $\longleftrightarrow$ $(\forall x\ y.\ sel\ (x,y) = f\ x\ y)$
⟨*proof*⟩

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*: $x \rightarrow^+_T y \Longrightarrow \neg\ y \rightarrow^+_T x$
⟨*proof*⟩

lemma *in-arcs-root*: *in-arcs T root* $= \{\}$
⟨*proof*⟩

lemma *dominated-not-root*: $u \rightarrow_T v \Longrightarrow v \neq root$
⟨*proof*⟩

lemma *dominated-notin-awalk*: $[\![u \rightarrow_T v;\ awalk\ r\ p\ u]\!] \Longrightarrow v \notin set\ (awalk\text{-}verts\ r\ p)$
⟨*proof*⟩

lemma *apath-if-awalk*: *awalk r p v* $\Longrightarrow$ *apath r p v*
⟨*proof*⟩

lemma *awalk-verts-arc1-app*: *tail T e* $\in$ *set* (*awalk-verts r* (*p1*@*e*#*p2*))
⟨*proof*⟩

lemma *apath-over-inarc-if-dominated*:
assumes $u \rightarrow_T v$
shows $\exists p.\ apath\ root\ p\ v \wedge u \in set\ (awalk\text{-}verts\ root\ p)$
⟨*proof*⟩

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*) ≤ *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*
  ⟨*proof*⟩

lemma *tailT-eq-tailG*: *tail T* = *tail G*
  ⟨*proof*⟩

lemma *headT-eq-headG*: *head T* = *head G*
  ⟨*proof*⟩

lemma *verts-T-subset-G*: *verts T* ⊆ *verts G*
  ⟨*proof*⟩

lemma *reachable-verts-G-subset-T*:
  ∀ *x*∈*verts G*. *root* →**G x* ⟹ *verts T* ⊇ *verts G*
  ⟨*proof*⟩

lemma *reachable-verts-G-eq-T*: ∀ *x*∈*verts G*. *root* →**G x* ⟹ *verts T* = *verts G*
  ⟨*proof*⟩

lemma *connected-verts-G-eq-T*:
  assumes *graph G* **and** *connected G*
  shows *verts T* = *verts G*
⟨*proof*⟩

lemma *Suc-card-if-fin*: *fin-digraph G* ⟹ ∃ *n*. *Suc n* = *card* (*verts G*)
  ⟨*proof*⟩

corollary *Suc-card-if-graph*: *graph G* ⟹ ∃ *n*. *Suc n* = *card* (*verts G*)

⟨*proof*⟩

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

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)}
⟨*proof*⟩

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*)}
⟨*proof*⟩

lemma *fin-verts-T*: *fin-digraph G* ⟹ *finite* (*verts T*)
⟨*proof*⟩

corollary *fin-verts-T′*: *graph G* ⟹ *finite* (*verts T*)
⟨*proof*⟩

lemma *fin-arcs-T*: *fin-digraph G* ⟹ *finite* (*arcs T*)
⟨*proof*⟩

corollary *fin-arcs-T′*: *graph G* ⟹ *finite* (*arcs T*)

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *arcs-graph-G-ge-2vertsT*:
assumes *graph G*
shows *card* (*arcs G*) ≥ *2* ∗ (*card* (*verts T*) − *1*)
⟨*proof*⟩

lemma *arcs-graph-G-ge-2vertsG*:
⟦*graph G*; *connected G*⟧ ⟹ *card* (*arcs G*) ≥ *2* ∗ (*card* (*verts G*) − *1*)
⟨*proof*⟩

lemma *arcs-undir-G-eq-2vertsG*:
⟦*undirected-tree G*⟧ ⟹ *card* (*arcs G*) = *2* ∗ (*card* (*verts G*) − *1*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *source-no-inarc-T*: *head G e* = *root* ⟹ *e* ∉ *arcs T*
⟨*proof*⟩

lemma *source-all-outarcs-T*:
⟦*undirected-tree G*; *tail G e* = *root*; *e* ∈ *arcs G*⟧ ⟹ *e* ∈ *arcs T*
⟨*proof*⟩

lemma *cas-G-T*: *G.cas* = *cas*
⟨*proof*⟩

lemma *awalk-G-T*: *u* ∈ *verts T* ⟹ *set p* ⊆ *arcs T* ⟹ *G.awalk u p* = *awalk u p*
⟨*proof*⟩

corollary *awalk-G-T-root*: *set p* ⊆ *arcs T* ⟹ *G.awalk root p* = *awalk root p*
⟨*proof*⟩

lemma *awalk-verts-G-T*: $G.awalk\text{-}verts = awalk\text{-}verts$
⟨*proof*⟩

lemma *apath-sub-imp-apath*: $apath\ u\ p\ v \Longrightarrow G.apath\ u\ p\ v$
⟨*proof*⟩

lemma *outarc-inT-if-head-not-inarc*:
assumes *undirected-tree G*
and $tail\ G\ e2 = v$ **and** $e2 \in arcs\ G$ **and** $head\ G\ e2 \neq u$ **and** $u \rightarrow_T v$
shows $e2 \in arcs\ T$
⟨*proof*⟩

corollary *reverse-arc-if-out-arc-undir*:
$[\![undirected\text{-}tree\ G;\ tail\ G\ e2 = v;\ e2 \in arcs\ G;\ e2 \notin arcs\ T;\ u \rightarrow_T v]\!] \Longrightarrow head\ G\ e2 = u$
⟨*proof*⟩

lemma *undir-path-in-dir*:
assumes *undirected-tree G G.apath root p v*
shows $set\ p \subseteq arcs\ T$
⟨*proof*⟩

lemma *source-reach-all*: $[\![graph\ G;\ connected\ G;\ v \in verts\ G]\!] \Longrightarrow root \rightarrow^*_G v$
⟨*proof*⟩

lemma *apath-if-in-verts*: $[\![graph\ G;\ connected\ G;\ v \in verts\ G]\!] \Longrightarrow \exists p.\ G.apath\ root\ p\ v$
⟨*proof*⟩

lemma *undir-unique-awalk*: $[\![undirected\text{-}tree\ G;\ v \in verts\ G]\!] \Longrightarrow \exists! p.\ G.apath\ root\ p\ v$
⟨*proof*⟩

lemma *apath-in-dir-if-apath-G*:
assumes *undirected-tree G G.apath root p v*
shows *apath root p v*
⟨*proof*⟩

end

locale *bfs-locale* =
fixes *bfs* :: $('a, 'b)\ pre\text{-}digraph \Rightarrow 'a \Rightarrow ('a, 'b)\ pre\text{-}digraph$
assumes *bfs-correct*: $[\![wf\text{-}digraph\ G;\ r \in verts\ G;\ bfs\ G\ r = T]\!] \Longrightarrow bfs\text{-}tree\ G\ T\ r$

locale *undir-tree-todir* = *undirected-tree G* + *bfs-locale bfs*
for *G* :: $('a, 'b)\ pre\text{-}digraph$
and *bfs* :: $('a, 'b)\ pre\text{-}digraph \Rightarrow 'a \Rightarrow ('a, 'b)\ pre\text{-}digraph$
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*
⟨*proof*⟩

lemma *bfs-dir-tree-r*: *r* ∈ *verts G* ⟹ *bfs-tree G* (*dir-tree-r r*) *r*
⟨*proof*⟩

lemma *dir-tree-r-dom-in-G*: *r* ∈ *verts G* ⟹ *u* →$_{dir\text{-}tree\text{-}r\ r}$ *v* ⟹ *u* →$_{G}$ *v*
⟨*proof*⟩

lemma *verts-nempty*: *verts G* ≠ {}
⟨*proof*⟩

lemma *card-gt0*: *card* (*verts G*) > *0*
⟨*proof*⟩

lemma *Suc-card-1-eq-card*[*intro*]: *Suc* (*card* (*verts G*) − *1*) = *card* (*verts G*)
⟨*proof*⟩

lemma *verts-dir-tree-r-eq*[*simp*]: *r* ∈ *verts G* ⟹ *verts* (*dir-tree-r r*) = *verts G*
⟨*proof*⟩

lemma *tail-dir-tree-r-eq*: *r* ∈ *verts G* ⟹ *tail* (*dir-tree-r r*) *e* = *tail G e*
⟨*proof*⟩

lemma *head-dir-tree-r-eq*: *r* ∈ *verts G* ⟹ *head* (*dir-tree-r r*) *e* = *head G e*
⟨*proof*⟩

lemma *awalk-verts-G-T*: *r* ∈ *verts G* ⟹ *awalk-verts* = *pre-digraph.awalk-verts*
(*dir-tree-r r*)
⟨*proof*⟩

lemma *dir-tree-r-all-reach*: ⟦*r* ∈ *verts G*; *v* ∈ *verts G*⟧ ⟹ *r* →** *dir-tree-r r v*
⟨*proof*⟩

lemma *fin-verts-dir-tree-r-eq*: *r* ∈ *verts G* ⟹ *finite* (*verts* (*dir-tree-r r*))
⟨*proof*⟩

lemma *fin-arcs-dir-tree-r-eq*: *r* ∈ *verts G* ⟹ *finite* (*arcs* (*dir-tree-r r*))
⟨*proof*⟩

lemma *fin-directed-tree-r*: *r* ∈ *verts G* ⟹ *finite-directed-tree* (*dir-tree-r r*) *r*
⟨*proof*⟩

lemma *arcs-eq-2verts*: *card* (*arcs G*) = *2* ∗ (*card* (*verts G*) − *1*)
⟨*proof*⟩

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

lemma *unique-apath*: $[\![u \in verts\ G;\ v \in verts\ G]\!] \Longrightarrow \exists!p.\ apath\ u\ p\ v$
⟨*proof*⟩

lemma *apath-in-dir-if-apath-G*: $apath\ r\ p\ v \Longrightarrow pre\text{-}digraph.apath\ (dir\text{-}tree\text{-}r\ r)\ r\ p\ v$
⟨*proof*⟩

lemma *apath-verts-sub-awalk*:
$[\![apath\ u\ p1\ v;\ awalk\ u\ p2\ v]\!] \Longrightarrow set\ (awalk\text{-}verts\ u\ p1) \subseteq set\ (awalk\text{-}verts\ u\ p2)$
⟨*proof*⟩

lemma *dir-tree-arc1-in-apath*:
assumes $u \rightarrow_{dir\text{-}tree\text{-}r\ r} v$ **and** $r \in verts\ G$
shows $\exists\, p.\ apath\ r\ p\ v \wedge u \in set\ (awalk\text{-}verts\ r\ p)$
⟨*proof*⟩

lemma *dir-tree-arc1-in-awalk*:
$[\![u \rightarrow_{dir\text{-}tree\text{-}r\ r} v;\ r \in verts\ G;\ awalk\ r\ p\ v]\!] \Longrightarrow u \in set\ (awalk\text{-}verts\ r\ p)$
⟨*proof*⟩

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* :: $('a, 'b)\ pre\text{-}digraph \Rightarrow ('b \Rightarrow real) \Rightarrow 'a \Rightarrow nat \Rightarrow ('a, 'b)\ pre\text{-}digraph$
assumes *find-psp-tree*: $[\![r \in verts\ G;\ find\text{-}psp\text{-}tree\ G\ w\ r\ n = T]\!] \Longrightarrow psp\text{-}tree\ G\ T\ w\ r\ n$

context *psp-tree*
begin

lemma *dom-in-G*: $u \rightarrow_T v \Longrightarrow u \rightarrow_G v$
⟨*proof*⟩

lemma *tailT-eq-tailG*: $tail\ T = tail\ G$
⟨*proof*⟩

lemma *headT-eq-headG*: *head T = head G*
⟨*proof*⟩

lemma *verts-T-subset-G*: *verts T ⊆ verts G*
⟨*proof*⟩

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*⟩

lemma *reachable-verts-G-eq-T*:
⟦*fin-digraph G*; *∀ x∈verts G. source →*_G x*; *Suc n = card (verts G)*⟧ *⟹ verts T = verts G*
⟨*proof*⟩

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

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)*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)}
⟨*proof*⟩

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*)}
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *arcs-graph-G-ge-2vertsT*:
assumes *graph G*
shows *card* (*arcs G*) ≥ *2* ∗ (*card* (*verts T*) − *1*)
⟨*proof*⟩

lemma *arcs-graph-G-ge-2vertsG*:
⟦*graph G*; *connected G*; *Suc n* = *card* (*verts G*)⟧ ⟹ *card* (*arcs G*) ≥ *2* ∗ (*card* (*verts G*) − *1*)
⟨*proof*⟩

lemma *arcs-undir-G-eq-2vertsG*:
⟦*undirected-tree G*; *Suc n* = *card* (*verts G*)⟧ ⟹ *card* (*arcs G*) = *2* ∗ (*card* (*verts G*) − *1*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *source-no-inarc-T*: *head G e = source ⟹ e ∉ arcs T*
⟨*proof*⟩

lemma *source-all-outarcs-T*:
⟦*undirected-tree G*; *Suc n = card (verts G)*; *tail G e = source*; *e ∈ arcs G*⟧ ⟹ *e ∈ arcs T*
⟨*proof*⟩

lemma *cas-G-T*: *G.cas = cas*
⟨*proof*⟩

lemma *awalk-G-T*: *u ∈ verts T ⟹ set p ⊆ arcs T ⟹ G.awalk u p = awalk u p*
⟨*proof*⟩

corollary *awalk-G-T-root*: *set p ⊆ arcs T ⟹ G.awalk source p = awalk source p*
⟨*proof*⟩

lemma *awalk-verts-G-T*: *G.awalk-verts = awalk-verts*
⟨*proof*⟩

lemma *apath-sub-imp-apath*: *apath u p v ⟹ G.apath u p v*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

lemma *source-reach-all*: ⟦*graph G*; *connected G*; *v ∈ verts G*⟧ ⟹ *source →*$^*_G$ *v*
⟨*proof*⟩

lemma *apath-if-in-verts*: ⟦*graph G*; *connected G*; *v ∈ verts G*⟧ ⟹ ∃ *p*. *G.apath source p v*
⟨*proof*⟩

lemma *undir-unique-awalk*:

⟦*undirected-tree G*; *Suc n* = *card* (*verts G*); *v* ∈ *verts G*⟧ ⟹ ∃!*p*. *G.apath source p v*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *dir-tree-r-dom-in-G*: *r* ∈ *verts G* ⟹ *u* →$_{dir\text{-}tree\text{-}r\ r}$ *v* ⟹ *u* →$_{G}$ *v*
⟨*proof*⟩

lemma *verts-nempty*: *verts G* ≠ {}
⟨*proof*⟩

lemma *card-gt0*: *card* (*verts G*) > *0*
⟨*proof*⟩

lemma *Suc-card-1-eq-card*[*intro*]: *Suc* (*card* (*verts G*) − *1*) = *card* (*verts G*)
⟨*proof*⟩

lemma *verts-dir-tree-r-eq*[*simp*]: *r* ∈ *verts G* ⟹ *verts* (*dir-tree-r r*) = *verts G*
⟨*proof*⟩

lemma *tail-dir-tree-r-eq*: *r* ∈ *verts G* ⟹ *tail* (*dir-tree-r r*) *e* = *tail G e*
⟨*proof*⟩

lemma *head-dir-tree-r-eq*: *r* ∈ *verts G* ⟹ *head* (*dir-tree-r r*) *e* = *head G e*
⟨*proof*⟩

lemma *awalk-verts-G-T*: *r* ∈ *verts G* ⟹ *awalk-verts* = *pre-digraph.awalk-verts*

(*dir-tree-r r*)
⟨*proof*⟩

lemma *dir-tree-r-all-reach*: ⟦*r* ∈ *verts G*; *v* ∈ *verts G*⟧ ⟹ *r* →**dir-tree-r r v*
⟨*proof*⟩

lemma *fin-verts-dir-tree-r-eq*: *r* ∈ *verts G* ⟹ *finite* (*verts* (*dir-tree-r r*))
⟨*proof*⟩

lemma *fin-arcs-dir-tree-r-eq*: *r* ∈ *verts G* ⟹ *finite* (*arcs* (*dir-tree-r r*))
⟨*proof*⟩

lemma *fin-directed-tree-r*: *r* ∈ *verts G* ⟹ *finite-directed-tree* (*dir-tree-r r*) *r*
⟨*proof*⟩

lemma *arcs-eq-2verts*: *card* (*arcs G*) = *2* ∗ (*card* (*verts G*) − *1*)
⟨*proof*⟩

lemma *arcs-compl-un-eq-arcs*:
r ∈ *verts G* ⟹
{*e2* ∈ *arcs G*. ∃ *e1*∈*arcs* (*dir-tree-r r*). *head G e2* = *tail G e1* ∧ *head G e1* = *tail G e2*}
∪ *arcs* (*dir-tree-r r*) = *arcs G*
⟨*proof*⟩

lemma *unique-apath*: ⟦*u* ∈ *verts G*; *v* ∈ *verts G*⟧ ⟹ ∃!*p*. *apath u p v*
⟨*proof*⟩

lemma *apath-in-dir-if-apath-G*: *apath r p v* ⟹ *pre-digraph.apath* (*dir-tree-r r*) *r p v*
⟨*proof*⟩

lemma *apath-verts-sub-awalk*:
⟦*apath u p1 v*; *awalk u p2 v*⟧ ⟹ *set* (*awalk-verts u p1*) ⊆ *set* (*awalk-verts u p2*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

corollary *dir-tree-single*: *directed-tree* (|*verts* = {*r*}, *arcs* = {}, *tail* = *t*, *head* = *h*|) *r*
⟨*proof*⟩

lemma *split-list-not-last*: ⟦*y* ∈ *set xs*; *y* ≠ *last xs*⟧ ⟹ ∃ *as bs*. *as* @ *y* # *bs* = *xs* ∧ *bs* ≠ []
⟨*proof*⟩

lemma *split-last-eq*: ⟦*as* @ *y* # *bs* = *xs*; *bs* ≠ []⟧ ⟹ *last bs* = *last xs*
⟨*proof*⟩

lemma *split-list-last-sep*: ⟦*y* ∈ *set xs*; *y* ≠ *last xs*⟧ ⟹ ∃ *as bs*. *as* @ *y* # *bs* @ [*last xs*] = *xs*
⟨*proof*⟩

context *directed-tree*
begin

lemma *root-if-all-reach*: ∀ *v* ∈ *verts T*. *x* →**T *v* ⟹ *x* = *root*
⟨*proof*⟩

lemma *add-leaf-cas-preserv*:
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** *set p* ⊆ *arcs T* **and** *cas x p y*
shows *pre-digraph.cas T' x p y*
⟨*proof*⟩

lemma *add-leaf-awalk-preserv*:
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** *awalk x p y*
shows *pre-digraph.awalk T' x p y*
⟨*proof*⟩

lemma *add-leaf-awalk-T*:
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** *x* ∈ *verts T*
shows ∃ *p*. *pre-digraph.awalk T' root p x*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *cas-append-if*:
⟦*cas x ps u*; *tail G p* = *u*; *head G p* = *v*⟧ ⟹ *cas x* (*ps*@[*p*]) *v*
⟨*proof*⟩

lemma *add-leaf-awalk-T-new*:
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*
shows ∃*p*. *pre-digraph.awalk T' root p v*
⟨*proof*⟩

lemma *add-leaf-cas-orig*:
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** *set p* ⊆ *arcs T* **and** *pre-digraph.cas T' x p y*
shows *cas x p y*
⟨*proof*⟩

lemma *add-leaf-awalk-orig-aux*:
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** *x* ∈ *verts T* **and** *set p* ⊆ *arcs T* **and** *pre-digraph.awalk T' x p y*
shows *awalk x p y*
⟨*proof*⟩

lemma *add-leaf-cas-xT-if-yT*:
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 *u* ∈ *verts T* **and** *y* ∈ *verts T* **and** *set p* ⊆ *arcs T'* **and** *pre-digraph.cas T' x p y*
shows *x* ∈ *verts T*
⟨*proof*⟩

lemma *add-leaf-cas-xT-arcsT-if-yT*:
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** *y* ∈ *verts T* **and** *set p* ⊆ *arcs T'* **and** *pre-digraph.cas T' x p y*
shows *set p* ⊆ *arcs T* **and** *x* ∈ *verts T*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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 $\exists as.\ as\ @\ [a] = p$
⟨*proof*⟩

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** $u \in verts\ T$ **and** $p \neq []$ **and** *pre-digraph.awalk* $T'\ x\ p\ v$
shows $\exists as.\ as\ @\ [a] = p \wedge$ *pre-digraph.awalk* $T'\ x\ as\ u$
⟨*proof*⟩

lemma *add-leaf-awalk-new-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** $u \in verts\ T$ **and** $v \notin verts\ T$
and *pre-digraph.awalk* T' *root ps v* **and** *pre-digraph.awalk* T' *root es v*
shows *es = ps*
⟨*proof*⟩

lemma *add-leaf-awalk-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** $u \in verts\ T$ **and** $v \notin verts\ T$ **and** $x \in verts\ T'$
shows $\exists!p.$ *pre-digraph.awalk* T' *root p x*

⟨*proof*⟩

lemma *add-leaf-dir-tree*:
⟦$a \notin$ *arcs* T; $u \in$ *verts* T; $v \notin$ *verts* T⟧
⟹ *directed-tree* (|*verts* = *verts* $T \cup \{v\}$, *arcs* = *arcs* $T \cup \{a\}$,
tail = (*tail* T)($a := u$), *head* = (*head* T)($a := v$)|) *root*
⟨*proof*⟩

lemma *add-leaf-dom-preserv*:
⟦$a \notin$ *arcs* T; $x \to_T y$⟧
⟹ $x \to$(|*verts* = *verts* $T \cup \{v\}$, *arcs* = *arcs* $T \cup \{a\}$, *tail* = (*tail* T)($a := u$), *head* = (*head* T)($a := v$
y
⟨*proof*⟩

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*⟩

lemma *reachable-via-child-impl-same*:
assumes $x \to^*{}_T v$ **and** $y \to^*{}_T v$ **and** $u \to_T x$ **and** $u \to_T y$
shows $x = y$
⟨*proof*⟩

lemma *new-leaf-last-in-orig-if-arcs-in-orig*:
assumes $x \to^*{}_T y$
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*⟩

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 $\exists\, z \in last\text{-}branching\text{-}points.\ x \rightarrow^*_T z$
⟨*proof*⟩

lemma *subgraph-no-last-branch-chain*:
assumes *subgraph* $C\ T$
and *finite* $(verts\ T)$
and $verts\ C \subseteq verts\ T - \{x.\ \exists\, y \in last\text{-}branching\text{-}points.\ x \rightarrow^*_T y\}$
shows *wf-digraph.is-chain* C
⟨*proof*⟩

lemma *reach-from-last-in-chain*:
assumes $\exists\, y \in last\text{-}branching\text{-}points.\ y \rightarrow^+_T x$
shows $x \in verts\ T - \{x.\ \exists\, y \in last\text{-}branching\text{-}points.\ x \rightarrow^*_T y\}$
⟨*proof*⟩

Directed Trees don't have merging points.

lemma *merging-empty*: *merging-points* $= \{\}$
⟨*proof*⟩

lemma *subgraph-no-last-merge-chain*:
assumes *subgraph* $C\ T$
shows *wf-digraph.is-chain'* C
⟨*proof*⟩

6.4 Converting to Trees of Lists

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

lemma *to-list-tree-union-verts-eq*: $\bigcup(set\ `\ verts\ to\text{-}list\text{-}tree) = verts\ T$
⟨*proof*⟩

lemma *to-list-tree-cas*: $cas\ u\ p\ v \longleftrightarrow pre\text{-}digraph.cas\ to\text{-}list\text{-}tree\ [u]\ p\ [v]$
⟨*proof*⟩

lemma *to-list-tree-awalk*: *awalk u p v* $\longleftrightarrow$ *pre-digraph.awalk to-list-tree* [*u*] *p* [*v*]
⟨*proof*⟩

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*⟩

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*⟩

lemma *to-list-tree-directed-tree*: *directed-tree to-list-tree* [*root*]
⟨*proof*⟩

lemma *to-list-tree-disjoint-verts*:
⟦*u* ∈ *verts to-list-tree*; *v* ∈ *verts to-list-tree*; *u*≠*v*⟧ ⟹ *set u* ∩ *set v* = {}
⟨*proof*⟩

lemma *to-list-tree-nempty*: *v* ∈ *verts to-list-tree* ⟹ *v* ≠ []
⟨*proof*⟩

lemma *to-list-tree-single*: *v* ∈ *verts to-list-tree* ⟹ ∃ *x*. *v* = [*x*] ∧ *x* ∈ *verts T*
⟨*proof*⟩

lemma *to-list-tree-dom-iff*: *x* →$_T$ *y* $\longleftrightarrow$ [*x*] →$_{to\text{-}list\text{-}tree}$ [*y*]
⟨*proof*⟩

end

locale *fin-list-directed-tree* = *finite-directed-tree T* **for** *T* :: ('*a list*,'*b*) *pre-digraph*
+
assumes *disjoint-verts*: ⟦*u* ∈ *verts T*; *v* ∈ *verts T*; *u* ≠ *v*⟧ ⟹ *set u* ∩ *set v* = {}
and *nempty-verts*: *v* ∈ *verts T* ⟹ *v* ≠ []

context *finite-directed-tree*
begin

lemma *to-list-tree-fin-digraph*: *fin-digraph to-list-tree*
⟨*proof*⟩

lemma *to-list-tree-finite-directed-tree*: *finite-directed-tree to-list-tree* [*root*]
⟨*proof*⟩

lemma *to-list-tree-fin-list-directed-tree*: *fin-list-directed-tree* [*root*] *to-list-tree*

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dtree-size-decr-aux*:
assumes (*x*,*y*) ∈ *fset xs*
shows *size x* < *size* (*Node r xs*)
⟨*proof*⟩

lemma *dtree-size-decr-aux'*: *t1* ∈ *fst* ‘ *fset xs* ⟹ *size t1* < *size* (*Node r xs*)
⟨*proof*⟩

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*⟩

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*)

⟨*proof*⟩

lemma *finite-darcs*: *finite* (*darcs t*)
⟨*proof*⟩

lemma *dverts-child-subseteq*: $x \in fst\ `\ fset\ xs \Longrightarrow dverts\ x \subseteq dverts\ (Node\ r\ xs)$
⟨*proof*⟩

lemma *dverts-suc-subseteq*: $x \in fst\ `\ fset\ (sucs\ t) \Longrightarrow dverts\ x \subseteq dverts\ t$
⟨*proof*⟩

lemma *dverts-root-or-child*: $v \in dverts\ (Node\ r\ xs) \Longrightarrow v = r \vee v \in (\bigcup (t,e) \in fset\ xs.\ dverts\ t)$
⟨*proof*⟩

lemma *dverts-root-or-suc*: $v \in dverts\ t \Longrightarrow v = root\ t \vee (\exists\,(t,e) \in fset\ (sucs\ t).v \in dverts\ t)$
⟨*proof*⟩

lemma *dverts-child-if-not-root*:
$[\![v \in dverts\ (Node\ r\ xs);\ v \neq r]\!] \Longrightarrow \exists\, t{\in}fst\ `\ fset\ xs.\ v \in dverts\ t$
⟨*proof*⟩

lemma *dverts-suc-if-not-root*:
$[\![v \in dverts\ t;\ v \neq root\ t]\!] \Longrightarrow \exists\, t{\in}fst\ `\ fset\ (sucs\ t).\ v \in dverts\ t$
⟨*proof*⟩

lemma *darcs-child-subseteq*: $x \in fst\ `\ fset\ xs \Longrightarrow darcs\ x \subseteq darcs\ (Node\ r\ xs)$
⟨*proof*⟩

lemma *mset-sum-elem*: $x \in\# (\sum y \in fset\ Y.\ f\ y) \Longrightarrow \exists\, y \in fset\ Y.\ x \in\# f\ y$
⟨*proof*⟩

lemma *mset-sum-elem-iff*: $x \in\# (\sum y \in fset\ Y.\ f\ y) \longleftrightarrow (\exists\, y \in fset\ Y.\ x \in\# f\ y)$
⟨*proof*⟩

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

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

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

lemma *darcs-mset-suc-if-nsnd*:
⟦*x* ∈# *darcs-mset t*; *x* ∉ *snd* ‘ *fset* (*sucs t*)⟧ ⟹ ∃ (*t1*,*e1*) ∈ *fset* (*sucs t*). *x* ∈# *darcs-mset t1*
⟨*proof*⟩

lemma *darcs-mset-if-nchild*:
⟦*x* ∈# *darcs-mset* (*Node r xs*); ∄ *t1 e1*. (*t1*,*e1*) ∈ *fset xs* ∧ *x* ∈# *darcs-mset t1*⟧
⟹ *x* ∈ *snd* ‘ *fset xs*
⟨*proof*⟩

lemma *darcs-mset-if-nsuc*:
⟦*x* ∈# *darcs-mset t*; ∄ *t1 e1*. (*t1*,*e1*) ∈ *fset* (*sucs t*) ∧ *x* ∈# *darcs-mset t1*⟧
⟹ *x* ∈ *snd* ‘ *fset* (*sucs t*)
⟨*proof*⟩

lemma *darcs-mset-if-snd*[*intro*]: *x* ∈ *snd* ‘ *fset xs* ⟹ *x* ∈# *darcs-mset* (*Node r xs*)
⟨*proof*⟩

lemma *darcs-mset-suc-if-snd*[*intro*]: *x* ∈ *snd* ‘ *fset* (*sucs t*) ⟹ *x* ∈# *darcs-mset t*
⟨*proof*⟩

lemma *darcs-mset-if-child*[*intro*]:
⟦(*t1*,*e1*) ∈ *fset xs*; *x* ∈# *darcs-mset t1*⟧ ⟹ *x* ∈# *darcs-mset* (*Node r xs*)
⟨*proof*⟩

lemma *darcs-mset-if-suc*[*intro*]:
⟦(*t1*,*e1*) ∈ *fset* (*sucs t*); *x* ∈# *darcs-mset t1*⟧ ⟹ *x* ∈# *darcs-mset t*
⟨*proof*⟩

lemma *darcs-mset-sub-darcs*: *set-mset* (*darcs-mset t*) ⊆ *darcs t*
⟨*proof*⟩

lemma *darcs-sub-darcs-mset*: *darcs t* ⊆ *set-mset* (*darcs-mset t*)
⟨*proof*⟩

lemma *darcs-mset-eq-darcs*[*simp*]: *set-mset* (*darcs-mset t*) = *darcs t*
⟨*proof*⟩

lemma *dverts-mset-elem*:
x ∈# *dverts-mset* (*Node r xs*) ⟹ (∃ (*t*,*e*) ∈ *fset xs*. *x* ∈# *dverts-mset t*) ∨ *x* = *r*
⟨*proof*⟩

lemma *dverts-mset-if-nroot*:
⟦*x* ∈# *dverts-mset* (*Node r xs*); *x* ≠ *r*⟧ ⟹ ∃ (*t1*,*e1*) ∈ *fset xs*. *x* ∈# *dverts-mset t1*
⟨*proof*⟩

lemma *dverts-mset-suc-if-nroot*:
⟦$x \in\#$ *dverts-mset t*; $x \neq$ *root t*⟧ $\Longrightarrow \exists\,(t1,e1) \in$ *fset* (*sucs t*). $x \in\#$ *dverts-mset t1*
⟨*proof*⟩

lemma *dverts-mset-if-nchild*:
⟦$x \in\#$ *dverts-mset* (*Node r xs*); $\nexists\, t1\ e1.\ (t1,e1) \in$ *fset xs* $\wedge\ x \in\#$ *dverts-mset t1*⟧ $\Longrightarrow x = r$
⟨*proof*⟩

lemma *dverts-mset-if-nsuc*:
⟦$x \in\#$ *dverts-mset t*; $\nexists\, t1\ e1.\ (t1,e1) \in$ *fset* (*sucs t*) $\wedge\ x \in\#$ *dverts-mset t1*⟧ $\Longrightarrow$ $x =$ *root t*
⟨*proof*⟩

lemma *dverts-mset-if-root*[*intro*]: $x = r \Longrightarrow x \in\#$ *dverts-mset* (*Node r xs*)
⟨*proof*⟩

lemma *dverts-mset-suc-if-root*[*intro*]: $x =$ *root t* $\Longrightarrow x \in\#$ *dverts-mset t*
⟨*proof*⟩

lemma *dverts-mset-if-child*[*intro*]:
⟦(*t1*,*e1*) $\in$ *fset xs*; $x \in\#$ *dverts-mset t1*⟧ $\Longrightarrow x \in\#$ *dverts-mset* (*Node r xs*)
⟨*proof*⟩

lemma *dverts-mset-if-suc*[*intro*]:
⟦(*t1*,*e1*) $\in$ *fset* (*sucs t*); $x \in\#$ *dverts-mset t1*⟧ $\Longrightarrow x \in\#$ *dverts-mset t*
⟨*proof*⟩

lemma *dverts-mset-sub-dverts*: *set-mset* (*dverts-mset t*) $\subseteq$ *dverts t*
⟨*proof*⟩

lemma *dverts-sub-dverts-mset*: *dverts t* $\subseteq$ *set-mset* (*dverts-mset t*)
⟨*proof*⟩

lemma *dverts-mset-eq-dverts*[*simp*]: *set-mset* (*dverts-mset t*) = *dverts t*
⟨*proof*⟩

lemma *mset-sum-count-le*: $y \in$ *fset Y* $\Longrightarrow$ *count* (*f y*) *x* $\leq$ *count* ($\sum y \in$ *fset Y*. *f y*) *x*
⟨*proof*⟩

lemma *darcs-mset-alt*:
darcs-mset (*Node r xs*) = ($\sum$ (*t*,*e*) $\in$ *fset xs*. {#*e*#}) + ($\sum$ (*t*,*e*) $\in$ *fset xs*. *darcs-mset t*)
⟨*proof*⟩

lemma *darcs-mset-ge-child*:
t1 $\in$ *fst* ‘ *fset xs* $\Longrightarrow$ *count* (*darcs-mset t1*) *x* $\leq$ *count* (*darcs-mset* (*Node r xs*)) *x*

⟨*proof*⟩

lemma *darcs-mset-ge-suc*:
t1 ∈ *fst* ‘ *fset* (*sucs t*) ⟹ *count* (*darcs-mset t1*) *x* ≤ *count* (*darcs-mset t*) *x*
⟨*proof*⟩

lemma *darcs-mset-count-sum-aux*:
(∑ (*t1*,*e1*) ∈ *fset xs*. *count* (*darcs-mset t1*) *x*) = *count* ((∑ (*t*,*e*) ∈ *fset xs*. *darcs-mset t*)) *x*
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *darcs-mset-count-sum-ge*:
(∑ (*t1*,*e1*) ∈ *fset xs*. *count* (*darcs-mset t1*) *x*) ≤ *count* (*darcs-mset* (*Node r xs*)) *x*
⟨*proof*⟩

lemma *wf-darcs-alt*: *wf-darcs t* ⟷ (∀ *x*. *count* (*darcs-mset t*) *x* ≤ *1*)
⟨*proof*⟩

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*}) = {}
⟨*proof*⟩

lemma *disjoint-darcs-single*: *e* ∉ *darcs t* ⟷ *disjoint-darcs* {|(*t*,*e*)|}
⟨*proof*⟩

lemma *disjoint-darcs-insert*: *disjoint-darcs* (*finsert x xs*) ⟹ *disjoint-darcs xs*
⟨*proof*⟩

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

lemma *disjoint-darcs-if-wf-aux1*: ⟦*wf-darcs* (*Node r xs*); (*t1*,*e1*) ∈ *fset xs*⟧ ⟹ *e1* ∉ *darcs t1*
⟨*proof*⟩

lemma *fset-sum-ge-elem2*:
⟦$x \in$ *fset* X; $y \in$ *fset* X; $x \neq y$⟧ $\Longrightarrow$ $(f :: 'a \Rightarrow nat)\ x + f\ y \leq (\sum x \in$ *fset* $X.\ f\ x)$
⟨*proof*⟩

lemma *darcs-children-count-ge2-aux*:
assumes (*t1*,*e1*) $\in$ *fset xs* **and** (*t2*,*e2*) $\in$ *fset xs* **and** (*t1*,*e1*) $\neq$ (*t2*,*e2*)
and $e \in$ *darcs t1* **and** $e \in$ *darcs t2*
shows $(\sum$ (*t1*, *e1*)$\in$*fset xs*. *count* (*darcs-mset t1*) $e) \geq 2$
⟨*proof*⟩

lemma *darcs-children-count-ge2*:
assumes (*t1*,*e1*) $\in$ *fset xs* **and** (*t2*,*e2*) $\in$ *fset xs* **and** (*t1*,*e1*) $\neq$ (*t2*,*e2*)
and $e \in$ *darcs t1* **and** $e \in$ *darcs t2*
shows *count* (*darcs-mset* (*Node r xs*)) $e \geq 2$
⟨*proof*⟩

lemma *darcs-children-count-not1*:
⟦(*t1*,*e1*) $\in$ *fset xs*; (*t2*,*e2*) $\in$ *fset xs*; (*t1*,*e1*) $\neq$ (*t2*,*e2*); $e \in$ *darcs t1*; $e \in$ *darcs t2*⟧
$\Longrightarrow$ *count* (*darcs-mset* (*Node r xs*)) $e \neq 1$
⟨*proof*⟩

lemma *disjoint-darcs-if-wf-aux2*:
assumes *wf-darcs* (*Node r xs*)
and (*t1*,*e1*) $\in$ *fset xs* **and** (*t2*,*e2*) $\in$ *fset xs* **and** (*t1*,*e1*) $\neq$ (*t2*,*e2*)
shows *darcs t1* $\cap$ *darcs t2* $= \{\}$
⟨*proof*⟩

lemma *darcs-child-count-ge1*:
⟦(*t1*,*e1*) $\in$ *fset xs*; *e2* $\in$ *darcs t1*⟧ $\Longrightarrow$ *count* $(\sum$ (*t*, *e*)$\in$*fset xs*. *darcs-mset t*) *e2* ≥ 1
⟨*proof*⟩

lemma *darcs-snd-count-ge1*:
(*t2*,*e2*) $\in$ *fset xs* $\Longrightarrow$ *count* $(\sum$ (*t*, *e*)$\in$*fset xs*. $\{\#e\#\})$ *e2* ≥ 1
⟨*proof*⟩

lemma *darcs-child-count-ge2*:
⟦(*t1*,*e1*) $\in$ *fset xs*; (*t2*,*e2*) $\in$ *fset xs*; *e2* $\in$ *darcs t1*⟧ $\Longrightarrow$ *count* (*darcs-mset* (*Node r xs*)) *e2* ≥ 2
⟨*proof*⟩

lemma *disjoint-darcs-if-wf-aux3*:
assumes *wf-darcs* (*Node r xs*) **and** (*t1*,*e1*) $\in$ *fset xs* **and** (*t2*,*e2*) $\in$ *fset xs*
shows *e2* $\notin$ *darcs t1*
⟨*proof*⟩

lemma *darcs-snds-count-ge2-aux*:

assumes $(t1,e1) \in fset\ xs$ **and** $(t2,e2) \in fset\ xs$ **and** $(t1,e1) \neq (t2,e2)$ **and** $e1 = e2$
shows $count\ (\sum (t,\ e) \in fset\ xs.\ \{\#e\#\})\ e2 \geq 2$
⟨*proof*⟩

lemma *darcs-snds-count-ge2*:
$[\![(t1,e1) \in fset\ xs;\ (t2,e2) \in fset\ xs;\ (t1,e1) \neq (t2,e2);\ e1 = e2]\!]$
$\Longrightarrow count\ (darcs\text{-}mset\ (Node\ r\ xs))\ e2 \geq 2$
⟨*proof*⟩

lemma *disjoint-darcs-if-wf-aux4*:
assumes *wf-darcs* $(Node\ r\ xs)$
and $(t1,e1) \in fset\ xs$
and $(t2,e2) \in fset\ xs$
and $(t1,e1) \neq (t2,e2)$
shows $e1 \neq e2$
⟨*proof*⟩

lemma *disjoint-darcs-if-wf-aux5*:
$[\![wf\text{-}darcs\ (Node\ r\ xs);\ (t1,e1) \in fset\ xs;\ (t2,e2) \in fset\ xs;\ (t1,e1) \neq (t2,e2)]\!]$
$\Longrightarrow (darcs\ t1 \cup \{e1\}) \cap (darcs\ t2 \cup \{e2\}) = \{\}$
⟨*proof*⟩

lemma *disjoint-darcs-if-wf-xs*: *wf-darcs* $(Node\ r\ xs) \Longrightarrow$ *disjoint-darcs xs*
⟨*proof*⟩

lemma *disjoint-darcs-if-wf*: *wf-darcs* $t \Longrightarrow$ *disjoint-darcs* $(sucs\ t)$
⟨*proof*⟩

lemma *wf-darcs'-if-darcs*: *wf-darcs* $t \Longrightarrow$ *wf-darcs'* t
⟨*proof*⟩

lemma *wf-darcs-if-darcs'-aux*:
$[\![\forall (x,e) \in fset\ xs.\ wf\text{-}darcs\ x;\ disjoint\text{-}darcs\ xs]\!] \Longrightarrow wf\text{-}darcs\ (Node\ r\ xs)$
⟨*proof*⟩

lemma *wf-darcs-if-darcs'*: *wf-darcs'* $t \Longrightarrow$ *wf-darcs* t
⟨*proof*⟩

corollary *wf-darcs-iff-darcs'*: *wf-darcs* $t \longleftrightarrow$ *wf-darcs'* t
⟨*proof*⟩

lemma *disjoint-darcs-subset*:
assumes *xs* $|\subseteq|$ *ys* **and** *disjoint-darcs ys*
shows *disjoint-darcs xs*
⟨*proof*⟩

lemma *disjoint-darcs-img*:
assumes *disjoint-darcs xs* **and** $\forall (t,e) \in fset\ xs.\ darcs\ (f\ t) \subseteq darcs\ t$

shows *disjoint-darcs* ((λ(*t*,*e*). (*f t*,*e*)) |‘| *xs*) (**is** *disjoint-darcs ?xs*)
⟨*proof*⟩

lemma *dverts-mset-count-sum-ge*:
(∑(*t1*,*e1*) ∈ *fset xs*. *count* (*dverts-mset t1*) *x*) ≤ *count* (*dverts-mset* (*Node r xs*)) *x*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *dverts-child-count-ge1*:
⟦(*t1*,*e1*) ∈ *fset xs*; *x* ∈ *dverts t1*⟧ ⟹ *count* (∑(*t*, *e*)∈*fset xs*. *dverts-mset t*) *x* ≥ *1*
⟨*proof*⟩

lemma *root-not-child-if-wf-dverts*: ⟦*wf-dverts* (*Node r xs*); (*t1*,*e1*) ∈ *fset xs*⟧ ⟹ *r* ∉ *dverts t1*
⟨*proof*⟩

lemma *root-not-child-if-wf-dverts′*: *wf-dverts* (*Node r xs*) ⟹ ∀(*t1*,*e1*) ∈ *fset xs*. *r* ∉ *dverts t1*

⟨*proof*⟩

lemma *dverts-mset-ge-child*:
t1 ∈ *fst* ‘ *fset xs* ⟹ *count* (*dverts-mset t1*) *x* ≤ *count* (*dverts-mset* (*Node r xs*)) *x*
⟨*proof*⟩

lemma *wf-dverts-rec*[*dest*]:
assumes *wf-dverts* (*Node r xs*) **and** *t1* ∈ *fst* ‘ *fset xs*
shows *wf-dverts t1*
⟨*proof*⟩

lemma *wf-dverts′-if-dverts*: *wf-dverts t* ⟹ *wf-dverts′ t*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *wf-dverts-if-dverts′*: *wf-dverts′ t* ⟹ *wf-dverts t*
⟨*proof*⟩

corollary *wf-dverts-iff-dverts′*: *wf-dverts t* ⟷ *wf-dverts′ t*
⟨*proof*⟩

lemma *wf-dverts-sub*:
assumes *xs* |⊆| *ys* **and** *wf-dverts* (*Node r ys*)
shows *wf-dverts* (*Node r xs*)
⟨*proof*⟩

lemma *count-subset-le*:
xs |⊆| *ys* ⟹ *count* (∑ *x* ∈ *fset xs*. *f x*) *a* ≤ *count* (∑ *x* ∈ *fset ys*. *f x*) *a*
⟨*proof*⟩

lemma *darcs-mset-count-le-subset*:
xs |⊆| *ys* ⟹ *count* (*darcs-mset* (*Node r′ xs*)) *x* ≤ *count* (*darcs-mset* (*Node r ys*)) *x*
⟨*proof*⟩

lemma *wf-darcs-sub*: ⟦*xs* |⊆| *ys*; *wf-darcs* (*Node r′ ys*)⟧ ⟹ *wf-darcs* (*Node r xs*)
⟨*proof*⟩

lemma *wf-darcs-sucs*: ⟦*wf-darcs t*; *x* ∈ *fset* (*sucs t*)⟧ ⟹ *wf-darcs* (*Node r* {|*x*|})
⟨*proof*⟩

lemma *size-fset-alt*:

size-fset (*size-prod snd* (λ-. *0*)) (*map-prod* (λ*t*. (*t*, *size t*)) (λ*x*. *x*) |‘| *xs*)
= (∑ (*x*,*y*)∈ *fset xs*. *size x* + *2*)
⟨*proof*⟩

lemma *dtree-size-alt*: *size* (*Node r xs*) = (∑ (*x*,*y*)∈ *fset xs*. *size x* + *2*) + *1*
⟨*proof*⟩

lemma *dtree-size-eq-root*: *size* (*Node r xs*) = *size* (*Node r′ xs*)
⟨*proof*⟩

lemma *size-combine-decr*: *size* (*Node* (*r*@*root t1*) (*sucs t1*)) < *size* (*Node r* {|(*t1*, *e1*)|})
⟨*proof*⟩

lemma *size-le-if-child-subset*: *xs* |⊆| *ys* ⟹ *size* (*Node r xs*) ≤ *size* (*Node v ys*)
⟨*proof*⟩

lemma *size-le-if-sucs-subset*: *sucs t1* |⊆| *sucs t2* ⟹ *size t1* ≤ *size t2*
⟨*proof*⟩

lemma *combine-uneq*: *Node r* {|(*t1*, *e1*)|} ≠ *Node* (*r*@*root t1*) (*sucs t1*)
⟨*proof*⟩

lemma *child-uneq*: *t* ∈ *fst* ‘ *fset xs* ⟹ *Node r xs* ≠ *t*
⟨*proof*⟩

lemma *suc-uneq*: *t1* ∈ *fst* ‘ *fset* (*sucs t*) ⟹ *t* ≠ *t1*
⟨*proof*⟩

lemma *singleton-uneq*: *Node r* {|(*t*,*e*)|} ≠ *t*
⟨*proof*⟩

lemma *child-uneq′*: *t* ∈ *fst* ‘ *fset xs* ⟹ *Node r xs* ≠ *Node v* (*sucs t*)
⟨*proof*⟩

lemma *suc-uneq′*: *t1* ∈ *fst* ‘ *fset* (*sucs t*) ⟹ *t* ≠ *Node v* (*sucs t1*)
⟨*proof*⟩

lemma *singleton-uneq′*: *Node r* {|(*t*,*e*)|} ≠ *Node v* (*sucs t*)
⟨*proof*⟩

lemma *singleton-suc*: *t* ∈ *fst* ‘ *fset* (*sucs* (*Node r* {|(*t*,*e*)|}))
⟨*proof*⟩

lemma *fcard-image-le*: *fcard* (*f* |‘| *xs*) ≤ *fcard xs*
⟨*proof*⟩

lemma *sum-img-le*:
assumes ∀ *t* ∈ *fst* ‘ *fset xs*. (*g*::′*a* ⇒ *nat*) (*f t*) ≤ *g t*

shows $(\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs).\ g\ x) \leq (\sum (x,y) \in fset\ xs.\ g\ x)$
⟨*proof*⟩

lemma *dtree-size-img-le*:
assumes $\forall\, t \in fst\ `\ fset\ xs.\ size\ (f\ t) \leq size\ t$
shows $size\ (Node\ r\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs)) \leq size\ (Node\ r\ xs)$
⟨*proof*⟩

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 $(\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs).\ g\ x) < (\sum (x,y) \in fset\ xs.\ g\ x)$
⟨*proof*⟩

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*⟩

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 $(\sum (x,y) \in fset\ ((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs).\ g\ x) = (\sum (x,y) \in fset\ xs.\ g\ x)$
⟨*proof*⟩

lemma *elem-neq-if-fset-neq*:
$((\lambda(t,e).\ (f\ t,\ e))\ |`|\ xs) \neq xs \Longrightarrow \exists\, t \in fst\ `\ fset\ xs.\ f\ t \neq t$
⟨*proof*⟩

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*⟩

lemma *ffold-eq-fold*: $[\![finite\ xs;\ f = g]\!] \Longrightarrow ffold\ f\ acc\ (Abs\text{-}fset\ xs) = Finite\text{-}Set.fold\ g\ acc\ xs$
⟨*proof*⟩

lemma *Abs-fset-sub-if-sub*:
assumes *finite ys* **and** $xs \subseteq ys$
shows *Abs-fset xs* $|\subseteq|$ *Abs-fset ys*
⟨*proof*⟩

lemma *fold-commute-supset*:

assumes *finite ys* **and** $xs \subseteq ys$ **and** *P ys* **and** $\bigwedge ys\ xs.\ [\![xs \subseteq ys;\ P\ ys]\!] \Longrightarrow P\ xs$
and $\bigwedge xs.$ *comp-fun-commute* $(\lambda a\ b.$ *if* $a \notin xs \vee \neg Q\ a\ b \vee \neg P\ xs$ *then b else R a b*)
shows *Finite-Set.fold* $(\lambda a\ b.$ *if* $a \notin ys \vee \neg Q\ a\ b \vee \neg P\ ys$ *then b else R a b*) *acc xs*
$=$ *Finite-Set.fold* $(\lambda a\ b.$ *if* $a \notin xs \vee \neg Q\ a\ b \vee \neg P\ xs$ *then b else R a b*) *acc xs*
⟨*proof*⟩

lemma *dtail-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* $\vee \neg$*wf-darcs* (*Node r xs*)
then b else dtail x def)
shows $(f\ y \circ f\ x)\ z = (f\ x \circ f\ y)\ z$
⟨*proof*⟩

lemma *dtail-commute*:
comp-fun-commute $(\lambda(x,e2)\ b.$ *if* $(x,e2) \notin$ *fset xs* $\vee\ e \notin$ *darcs x* $\vee \neg$*wf-darcs* (*Node r xs*)
then b else dtail x def)
⟨*proof*⟩

lemma *dtail-f-alt*:
assumes $P = (\lambda xs.$ *wf-darcs* (*Node r xs*))
and $Q = (\lambda(t1,e1)\ b.\ e \in$ *darcs t1*$)$
and $R = (\lambda(t1,e1)\ b.$ *dtail t1 def*$)$
shows $(\lambda(t1,e1)\ b.$ *if* $(t1,e1) \notin$ *fset xs* $\vee\ e \notin$ *darcs t1* $\vee \neg$*wf-darcs* (*Node r xs*)
then b else dtail t1 def)
$= (\lambda a\ b.$ *if* $a \notin$ *fset xs* $\vee \neg\ Q\ a\ b \vee \neg\ P\ xs$ *then b else R a b*)
⟨*proof*⟩

lemma *dtail-f-alt-commute*:
assumes $P = (\lambda xs.$ *wf-darcs* (*Node r xs*))
and $Q = (\lambda(t1,e1)\ b.\ e \in$ *darcs t1*$)$
and $R = (\lambda(t1,e1)\ b.$ *dtail t1 def*$)$
shows *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*)
⟨*proof*⟩

lemma *dtail-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* $\vee \neg$*wf-darcs* (*Node r ys*)
then b else dtail x def) *def xs*
$=$ *ffold* $(\lambda(x,e2)\ b.$ *if* $(x,e2) \notin$ *fset xs* $\vee\ e \notin$ *darcs x* $\vee \neg$*wf-darcs* (*Node r xs*)
then b else dtail x def) *def xs*
⟨*proof*⟩

lemma *dtail-in-child-eq-child-ffold*:
assumes $(t,e1) \in fset\ xs$ **and** $e \in darcs\ t$ **and** *wf-darcs* $(Node\ r\ xs)$
shows *ffold* $(\lambda(x,e2)\ b.\ if\ (x,e2) \notin fset\ xs \vee e \notin darcs\ x \vee \neg$*wf-darcs* $(Node\ r\ xs)$
then b else dtail x def) *def xs*
$= dtail\ t\ def$
⟨*proof*⟩

lemma *dtail-in-child-eq-child*:
assumes $(t,e1) \in fset\ xs$ **and** $e \in darcs\ t$ **and** *wf-darcs* $(Node\ r\ xs)$
shows *dtail* $(Node\ r\ xs)\ def\ e = dtail\ t\ def\ e$
⟨*proof*⟩

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-darcs* $(Node\ r\ ys)$
then b else dtail x def) *def xs* $= def$
⟨*proof*⟩

lemma *dtail-notelem-eq-def*:
assumes $e \notin darcs\ t$
shows $dtail\ t\ def\ e = def\ e$
⟨*proof*⟩

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-darcs* $(Node\ r\ xs)$
then b else if e=e2 then root x else dhead x def e)
shows $(f\ y \circ f\ x)\ z = (f\ x \circ f\ y)\ z$
⟨*proof*⟩

lemma *dhead-commute*:
comp-fun-commute $(\lambda(x,e2)\ b.\ if\ (x,e2) \notin fset\ xs \vee e \notin (darcs\ x \cup \{e2\}) \vee \neg$*wf-darcs* $(Node\ r\ xs)$
then b else if e=e2 then root x else dhead x def e)
⟨*proof*⟩

lemma *dhead-ffold-f-alt*:
assumes $P = (\lambda xs.$ *wf-darcs* $(Node\ r\ xs))$ **and** $Q = (\lambda(x,e2)\ \text{-}.\ e \in (darcs\ x \cup \{e2\}))$
and $R = (\lambda(x,e2)\ \text{-}.$ *if e=e2 then root x else dhead x def e*)
shows $(\lambda(x,e2)\ b.\ if\ (x,e2) \notin fset\ xs \vee e \notin (darcs\ x \cup \{e2\}) \vee \neg$*wf-darcs* $(Node\ r\ xs)\ then\ b$
else if e=e2 then root x else dhead x def e)
$= (\lambda a\ b.\ if\ a \notin fset\ xs \vee \neg\ Q\ a\ b \vee \neg\ P\ xs\ then\ b\ else\ R\ a\ b)$
⟨*proof*⟩

lemma *dhead-ffold-f-alt-commute*:
assumes $P = (\lambda xs.\ wf\text{-}darcs\ (Node\ r\ xs))$ **and** $Q = (\lambda(x,e2)\ \text{-}.\ e \in (darcs\ x \cup \{e2\}))$
and $R = (\lambda(x,e2)\ \text{-}.\ if\ e{=}e2\ then\ root\ x\ else\ dhead\ x\ def\ e)$
shows $comp\text{-}fun\text{-}commute\ (\lambda a\ b.\ if\ a \notin fset\ xs \vee \neg\ Q\ a\ b \vee \neg\ P\ xs\ then\ b\ else\ R\ a\ b)$
⟨*proof*⟩

lemma *dhead-ffold-supset*:
assumes $xs\ |\subseteq|\ ys$ **and** $wf\text{-}darcs\ (Node\ r\ ys)$
shows $ffold\ (\lambda(x,e2)\ b.\ if\ (x,e2) \notin fset\ ys \vee e \notin (darcs\ x \cup \{e2\}) \vee \neg wf\text{-}darcs\ (Node\ r\ ys)\ then\ b$
$else\ if\ e{=}e2\ then\ root\ x\ else\ dhead\ x\ def\ e)\ (def\ e)\ xs$
$= ffold\ (\lambda(x,e2)\ b.\ if\ (x,e2) \notin fset\ xs \vee e \notin (darcs\ x \cup \{e2\}) \vee \neg wf\text{-}darcs\ (Node\ r\ xs)\ then\ b$
$else\ if\ e{=}e2\ then\ root\ x\ else\ dhead\ x\ def\ e)\ (def\ e)\ xs$
(**is** *ffold ?f - - = ffold ?g - -*)
⟨*proof*⟩

lemma *dhead-in-child-eq-child-ffold*:
assumes $(t,e1) \in fset\ xs$ **and** $e \in darcs\ t$ **and** $wf\text{-}darcs\ (Node\ r\ xs)$
shows $ffold\ (\lambda(x,e2)\ b.\ if\ (x,e2) \notin fset\ xs \vee e \notin (darcs\ x \cup \{e2\}) \vee \neg wf\text{-}darcs\ (Node\ r\ xs)$
$then\ b\ else\ if\ e{=}e2\ then\ root\ x\ else\ dhead\ x\ def\ e)\ (def\ e)\ xs$
$= dhead\ t\ def\ e$
⟨*proof*⟩

lemma *dhead-in-child-eq-child*:
assumes $(t,e1) \in fset\ xs$ **and** $e \in darcs\ t$ **and** $wf\text{-}darcs\ (Node\ r\ xs)$
shows $dhead\ (Node\ r\ xs)\ def\ e = dhead\ t\ def\ e$
⟨*proof*⟩

lemma *dhead-ffold-notelem-eq-def*:
assumes $\forall (t,e1) \in fset\ xs.\ e \notin darcs\ t \wedge e \neq e1$
shows $ffold\ (\lambda(x,e2)\ b.\ if\ (x,e2) \notin fset\ ys \vee e \notin (darcs\ x \cup \{e2\}) \vee \neg wf\text{-}darcs\ (Node\ r\ ys)\ then\ b$
$else\ if\ e{=}e2\ then\ root\ x\ else\ dhead\ x\ def\ e)\ (def\ e)\ xs = def\ e$
⟨*proof*⟩

lemma *dhead-notelem-eq-def*:
assumes $e \notin darcs\ t$
shows $dhead\ t\ def\ e = def\ e$
⟨*proof*⟩

lemma *dhead-in-set-eq-root-ffold*:
assumes $(t,e) \in fset\ xs$ **and** $wf\text{-}darcs\ (Node\ r\ xs)$
shows $ffold\ (\lambda(x,e2)\ b.\ if\ (x,e2) \notin fset\ xs \vee e \notin (darcs\ x \cup \{e2\}) \vee \neg wf\text{-}darcs\ (Node\ r\ xs)$
$then\ b\ else\ if\ e{=}e2\ then\ root\ x\ else\ dhead\ x\ def\ e)\ (def\ e)\ xs$

$= root\ t$ (**is** *ffold ?f′ - - = -*)
⟨*proof*⟩

lemma *dhead-in-set-eq-root*:
⟦$(t,e) \in fset\ xs$; *wf-darcs* $(Node\ r\ xs)$⟧ $\Longrightarrow$ *dhead* $(Node\ r\ xs)$ *def* $e = root\ t$
⟨*proof*⟩

lemma *self-subtree*: *is-subtree t t*
⟨*proof*⟩

lemma *subtree-trans*: *is-subtree x y* $\Longrightarrow$ *is-subtree y z* $\Longrightarrow$ *is-subtree x z*
⟨*proof*⟩

lemma *subtree-trans′*: *transp is-subtree*
⟨*proof*⟩

lemma *subtree-if-child*: $x \in fst\ `\ fset\ xs \Longrightarrow$ *is-subtree x* $(Node\ r\ xs)$
⟨*proof*⟩

lemma *subtree-if-suc*: *t1* $\in fst\ `\ fset\ (sucs\ t2) \Longrightarrow$ *is-subtree t1 t2*
⟨*proof*⟩

lemma *child-sub-if-strict-subtree*:
⟦*strict-subtree t1* $(Node\ r\ xs)$⟧ $\Longrightarrow \exists\, t3 \in fst\ `\ fset\ xs.$ *is-subtree t1 t3*
⟨*proof*⟩

lemma *suc-sub-if-strict-subtree*:
strict-subtree t1 t2 $\Longrightarrow \exists\, t3 \in fst\ `\ fset\ (sucs\ t2).$ *is-subtree t1 t3*
⟨*proof*⟩

lemma *subtree-size-decr*: ⟦*is-subtree t1 t2*; $t1 \neq t2$⟧ $\Longrightarrow$ *size t1* $<$ *size t2*
⟨*proof*⟩

lemma *subtree-size-decr′*: *strict-subtree t1 t2* $\Longrightarrow$ *size t1* $<$ *size t2*
⟨*proof*⟩

lemma *subtree-size-le*: *is-subtree t1 t2* $\Longrightarrow$ *size t1* $\leq$ *size t2*
⟨*proof*⟩

lemma *subtree-antisym*: ⟦*is-subtree t1 t2*; *is-subtree t2 t1*⟧ $\Longrightarrow$ *t1* $=$ *t2*
⟨*proof*⟩

lemma *subtree-antisym′*: *antisymp is-subtree*
⟨*proof*⟩

corollary *subtree-eq-if-trans-eq1*: ⟦*is-subtree t1 t2*; *is-subtree t2 t3*; *t1* $=$ *t3*⟧ $\Longrightarrow$
t1 $=$ *t2*
⟨*proof*⟩

corollary *subtree-eq-if-trans-eq2*: $[\![$*is-subtree t1 t2*; *is-subtree t2 t3*; *t1* = *t3*$]\!] \Longrightarrow$ *t2* = *t3*
⟨*proof*⟩

lemma *subtree-partial-ord*: *class.order is-subtree strict-subtree*
⟨*proof*⟩

lemma *finite-subtrees*: *finite* {*x*. *is-subtree x t*}
⟨*proof*⟩

lemma *subtrees-insert-union*:
{*x*. *is-subtree x* (*Node r xs*)} = *insert* (*Node r xs*) ($\bigcup$ *t1* ∈ *fst* ‘ *fset xs*. {*x*. *is-subtree x t1*})
⟨*proof*⟩

lemma *subtrees-insert-union-suc*:
{*x*. *is-subtree x t*} = *insert t* ($\bigcup$ *t1* ∈ *fst* ‘ *fset* (*sucs t*). {*x*. *is-subtree x t1*})
⟨*proof*⟩

lemma *darcs-subtree-subset*: *is-subtree x y* ⟹ *darcs x* ⊆ *darcs y*
⟨*proof*⟩

lemma *dverts-subtree-subset*: *is-subtree x y* ⟹ *dverts x* ⊆ *dverts y*
⟨*proof*⟩

lemma *single-subtree-root-dverts*:
is-subtree (*Node v2* {|(*t2*, *e2*)|}) *t1* ⟹ *v2* ∈ *dverts t1*
⟨*proof*⟩

lemma *single-subtree-child-root-dverts*:
is-subtree (*Node v2* {|(*t2*, *e2*)|}) *t1* ⟹ *root t2* ∈ *dverts t1*
⟨*proof*⟩

lemma *subtree-root-if-dverts*: *x* ∈ *dverts t* ⟹ ∃ *xs*. *is-subtree* (*Node x xs*) *t*
⟨*proof*⟩

lemma *subtree-child-if-strict-subtree*:
strict-subtree t1 t2 ⟹ ∃ *r xs*. *is-subtree* (*Node r xs*) *t2* ∧ *t1* ∈ *fst* ‘ *fset xs*
⟨*proof*⟩

lemma *subtree-child-if-dvert-notroot*:
assumes *v* ≠ *r* **and** *v* ∈ *dverts* (*Node r xs*)
shows ∃ *r′ ys zs*. *is-subtree* (*Node r′ ys*) (*Node r xs*) ∧ *Node v zs* ∈ *fst* ‘ *fset ys*
⟨*proof*⟩

lemma *subtree-child-if-dvert-notelem*:
$[\![$*v* ≠ *root t*; *v* ∈ *dverts t*$]\!] \Longrightarrow$ ∃ *r′ ys zs*. *is-subtree* (*Node r′ ys*) *t* ∧ *Node v zs* ∈ *fst* ‘ *fset ys*
⟨*proof*⟩

lemma *strict-subtree-subset*:
assumes *strict-subtree t* (*Node r xs*) **and** *xs* |⊆| *ys*
shows *strict-subtree t* (*Node r ys*)
⟨*proof*⟩

lemma *strict-subtree-singleton*:
⟦*strict-subtree t* (*Node r* {|*x*|}); *x* |∈| *xs*⟧
⟹ *strict-subtree t* (*Node r xs*)
⟨*proof*⟩

7.3.1 Finite Directed Trees to Dtree

context *finite-directed-tree*
begin

lemma *child-subtree*:
assumes *e* ∈ *out-arcs T r*
shows {*x*. (*head T e*) →** T x*} ⊆ {*x*. *r* →** T x*}
⟨*proof*⟩

lemma *child-strict-subtree*:
assumes *e* ∈ *out-arcs T r*
shows {*x*. (*head T e*) →** T x*} ⊂ {*x*. *r* →** T x*}
⟨*proof*⟩

lemma *child-card-decr*:
assumes *e* ∈ *out-arcs T r*
shows *Finite-Set.card* {*x*. (*head T e*) →** T x*} < *Finite-Set.card* {*x*. *r* →** T x*}
⟨*proof*⟩

function *to-dtree-aux* :: *'a* ⇒ (*'a*,*'b*) *dtree* **where**
to-dtree-aux r = *Node r* (*Abs-fset* {(*x*,*e*).
(*if e* ∈ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)})
⟨*proof*⟩
termination
⟨*proof*⟩

definition *to-dtree* :: (*'a*,*'b*) *dtree* **where**
to-dtree = *to-dtree-aux root*

abbreviation *from-dtree* :: (*'a*,*'b*) *dtree* ⇒ (*'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*
⟨*proof*⟩

lemma *verts-fset-id*: *fset* (*Abs-fset* (*verts T*)) = *verts T*
⟨*proof*⟩

lemma *arcs-fset-id*: *fset* (*Abs-fset* (*arcs T*)) = *arcs T*
⟨*proof*⟩

lemma *dtree-leaf-child-empty*:
leaf r ⟹ {(*x*,*e*). (*if e* ∈ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)} = {}
⟨*proof*⟩

lemma *dtree-leaf-no-children*: *leaf r* ⟹ *to-dtree-aux r* = *Node r* {||}
⟨*proof*⟩

lemma *dtree-children-alt*:
{(*x*,*e*). (*if e* ∈ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)}
= {(*x*,*e*). *e* ∈ *out-arcs T r* ∧ *x* = *to-dtree-aux* (*head T e*)}
⟨*proof*⟩

lemma *dtree-children-img-alt*:
(λ*e*. (*to-dtree-aux* (*head T e*),*e*)) ‘ (*out-arcs T r*)
= {(*x*,*e*). (*if e* ∈ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)}
⟨*proof*⟩

lemma *dtree-children-fin*:
finite {(*x*,*e*). (*if e* ∈ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)}
⟨*proof*⟩

lemma *dtree-children-fset-id*:
assumes *to-dtree-aux r* = *Node r xs*
shows *fset xs* = {(*x*,*e*). (*if e* ∈ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)}
⟨*proof*⟩

lemma *to-dtree-aux-empty-if-notT*:
assumes *r* ∉ *verts T*
shows *to-dtree-aux r* = *Node r* {||}
⟨*proof*⟩

lemma *to-dtree-aux-root*: *Dtree.root* (*to-dtree-aux r*) = *r*
⟨*proof*⟩

lemma *out-arc-if-child*:
assumes *x* ∈ (*fst* ‘ {(*x*,*e*). (*if e* ∈ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)})
shows ∃ *e*. *e* ∈ *out-arcs T r* ∧ *x* = *to-dtree-aux* (*head T e*)
⟨*proof*⟩

lemma *dominated-if-child-aux*:
assumes *x* ∈ (*fst* ‘ {(*x*,*e*). (*if e* ∈ *out-arcs T r then x* = *to-dtree-aux* (*head T e*) *else False*)})

shows $r \rightarrow_T (Dtree.root\ x)$
⟨*proof*⟩

lemma *dominated-if-child*:
⟦*to-dtree-aux r = Node r xs*; $x \in fst\ `\ fset\ xs$⟧ ⟹ $r \rightarrow_T (Dtree.root\ x)$
⟨*proof*⟩

lemma *image-add-snd-snd-id*: $snd\ `\ ((\lambda e.\ (to\text{-}dtree\text{-}aux\ (head\ T\ e), e))\ `\ x) = x$
⟨*proof*⟩

lemma *to-dtree-aux-child-in-verts*:
assumes *Node r' xs = to-dtree-aux r* **and** $x \in fst\ `\ fset\ xs$
shows $Dtree.root\ x \in verts\ T$
⟨*proof*⟩

lemma *to-dtree-aux-parent-in-verts*:
assumes *Node r' xs = to-dtree-aux r* **and** $x \in fst\ `\ fset\ xs$
shows $r \in verts\ T$
⟨*proof*⟩

lemma *dtree-out-arcs*:
$snd\ `\ \{(x,e).\ (if\ e \in out\text{-}arcs\ T\ r\ then\ x = to\text{-}dtree\text{-}aux\ (head\ T\ e)\ else\ False)\} = out\text{-}arcs\ T\ r$
⟨*proof*⟩

lemma *dtree-out-arcs-eq-snd*:
assumes *to-dtree-aux r = Node r xs*
shows $(snd\ `\ (fset\ xs)) = out\text{-}arcs\ T\ r$
⟨*proof*⟩

lemma *dtree-aux-fst-head-snd-aux*:
assumes $x \in \{(x,e).\ (if\ e \in out\text{-}arcs\ T\ r\ then\ x = to\text{-}dtree\text{-}aux\ (head\ T\ e)\ else\ False)\}$
shows $Dtree.root\ (fst\ x) = (head\ T\ (snd\ x))$
⟨*proof*⟩

lemma *dtree-aux-fst-head-snd*:
assumes *to-dtree-aux r = Node r xs* **and** $x \in fset\ xs$
shows $Dtree.root\ (fst\ x) = (head\ T\ (snd\ x))$
⟨*proof*⟩

lemma *child-if-dominated-aux*:
assumes $r \rightarrow_T x$
shows $\exists y \in (fst\ `\ \{(x,e).\ (if\ e \in out\text{-}arcs\ T\ r\ then\ x = to\text{-}dtree\text{-}aux\ (head\ T\ e)\ else\ False)\}).$
$Dtree.root\ y = x$
⟨*proof*⟩

lemma *child-if-dominated*:

assumes *to-dtree-aux r = Node r xs* **and** $r \rightarrow_T x$
shows $\exists\, y \in (fst\ `\ (fset\ xs)).\ Dtree.root\ y = x$
⟨*proof*⟩

lemma *to-dtree-aux-reach-in-dverts*: $[\![t = to\text{-}dtree\text{-}aux\ r;\ r \rightarrow^*_T x]\!] \Longrightarrow x \in dverts\ t$
⟨*proof*⟩

lemma *to-dtree-aux-dverts-reachable*:
$[\![t = to\text{-}dtree\text{-}aux\ r;\ x \in dverts\ t;\ r \in verts\ T]\!] \Longrightarrow r \rightarrow^*_T x$
⟨*proof*⟩

lemma *dverts-eq-reachable*: $r \in verts\ T \Longrightarrow dverts\ (to\text{-}dtree\text{-}aux\ r) = \{x.\ r \rightarrow^*_T x\}$
⟨*proof*⟩

lemma *dverts-eq-reachable'*: $[\![r \in verts\ T;\ t = to\text{-}dtree\text{-}aux\ r]\!] \Longrightarrow dverts\ t = \{x.\ r \rightarrow^*_T x\}$
⟨*proof*⟩

lemma *dverts-eq-verts*: *dverts to-dtree = verts T*
⟨*proof*⟩

lemma *arc-out-arc*: $e \in arcs\ T \Longrightarrow \exists\, v \in verts\ T.\ e \in out\text{-}arcs\ T\ v$
⟨*proof*⟩

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*⟩

lemma *darcs-in-arcs*: $e \in darcs\ to\text{-}dtree \Longrightarrow e \in arcs\ T$
⟨*proof*⟩

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*⟩

lemma *arcs-in-darcs*: $e \in arcs\ T \Longrightarrow e \in darcs\ to\text{-}dtree$
⟨*proof*⟩

lemma *darcs-eq-arcs*: *darcs to-dtree = arcs T*
⟨*proof*⟩

lemma *to-dtree-aux-self*:
assumes *Node r xs = to-dtree-aux r* **and** $(y,e) \in fset\ xs$
shows $y = to\text{-}dtree\text{-}aux\ (Dtree.root\ y)$
⟨*proof*⟩

lemma *to-dtree-aux-self-subtree*:

⟦*t1* = *to-dtree-aux r*; *is-subtree t2 t1*⟧ ⟹ *t2* = *to-dtree-aux* (*Dtree.root t2*)
⟨*proof*⟩

lemma *to-dtree-self-subtree*: *is-subtree t to-dtree* ⟹ *t* = *to-dtree-aux* (*Dtree.root t*)
⟨*proof*⟩

lemma *to-dtree-self-subtree′*: *is-subtree* (*Node r xs*) *to-dtree* ⟹ (*Node r xs*) = *to-dtree-aux r*
⟨*proof*⟩

lemma *child-if-dominated-to-dtree*:
⟦*is-subtree* (*Node r xs*) *to-dtree*; *r* →$_T$ *v*⟧ ⟹ ∃ *t*. *t* ∈ *fst* ‘ *fset xs* ∧ *Dtree.root t* = *v*
⟨*proof*⟩

lemma *child-if-dominated-to-dtree′*:
⟦*is-subtree* (*Node r xs*) *to-dtree*; *r* →$_T$ *v*⟧ ⟹ ∃ *ys*. *Node v ys* ∈ *fst* ‘ *fset xs*
⟨*proof*⟩

lemma *child-darc-tail-parent*:
assumes *Node r xs* = *to-dtree-aux r* **and** (*x*,*e*) ∈ *fset xs*
shows *tail T e* = *r*
⟨*proof*⟩

lemma *child-darc-head-root*:
⟦*Node r xs* = *to-dtree-aux r*; (*t*,*e*) ∈ *fset xs*⟧ ⟹ *head T e* = *Dtree.root t*
⟨*proof*⟩

lemma *child-darc-in-arcs*:
assumes *Node r xs* = *to-dtree-aux r* **and** (*x*,*e*) ∈ *fset xs*
shows *e* ∈ *arcs T*
⟨*proof*⟩

lemma *darcs-neq-if-dtrees-neq*:
⟦*Node r xs* = *to-dtree-aux r*; (*x*,*e1*) ∈ *fset xs*; (*y*,*e2*) ∈ *fset xs*; *x*≠*y*⟧ ⟹ *e1* ≠ *e2*
⟨*proof*⟩

lemma *dtrees-neq-if-darcs-neq*:
⟦*Node r xs* = *to-dtree-aux r*; (*x*,*e1*) ∈ *fset xs*; (*y*,*e2*) ∈ *fset xs*; *e1*≠*e2*⟧ ⟹ *x* ≠ *y*
⟨*proof*⟩

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*⟩

lemma *wf-dverts-to-dtree-aux1*: *r* ∉ *verts T* ⟹ *wf-dverts* (*to-dtree-aux r*)

⟨*proof*⟩

lemma *wf-dverts-to-dtree-aux2*: $r \in verts\ T \Longrightarrow t = to\text{-}dtree\text{-}aux\ r \Longrightarrow wf\text{-}dverts\ t$
⟨*proof*⟩

lemma *wf-dverts-to-dtree-aux*: *wf-dverts* (*to-dtree-aux r*)
⟨*proof*⟩

lemma *wf-dverts-to-dtree-aux'*: $t = to\text{-}dtree\text{-}aux\ r \Longrightarrow wf\text{-}dverts\ t$
⟨*proof*⟩

lemma *wf-dverts-to-dtree*: *wf-dverts to-dtree*
⟨*proof*⟩

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*⟩

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*⟩

lemma *wf-darcs-to-dtree-aux1*: $r \notin verts\ T \Longrightarrow wf\text{-}darcs\ (to\text{-}dtree\text{-}aux\ r)$
⟨*proof*⟩

lemma *wf-darcs-to-dtree-aux2*: $r \in verts\ T \Longrightarrow t = to\text{-}dtree\text{-}aux\ r \Longrightarrow wf\text{-}darcs\ t$
⟨*proof*⟩

lemma *wf-darcs-to-dtree-aux*: *wf-darcs* (*to-dtree-aux r*)
⟨*proof*⟩

lemma *wf-darcs-to-dtree-aux'*: $t = to\text{-}dtree\text{-}aux\ r \Longrightarrow wf\text{-}darcs\ t$
⟨*proof*⟩

lemma *wf-darcs-to-dtree*: *wf-darcs to-dtree*
⟨*proof*⟩

lemma *dtail-aux-elem-eq-tail*:
$t = to\text{-}dtree\text{-}aux\ r \Longrightarrow e \in darcs\ t \Longrightarrow dtail\ t\ def\ e = tail\ T\ e$
⟨*proof*⟩

lemma *dtail-elem-eq-tail*: $e \in darcs\ to\text{-}dtree \Longrightarrow dtail\ to\text{-}dtree\ def\ e = tail\ T\ e$
⟨*proof*⟩

lemma *to-dtree-dtail-eq-tail-aux*: *dtail to-dtree* (*tail T*) *e* = *tail T e*

⟨*proof*⟩

lemma *to-dtree-dtail-eq-tail*: *dtail to-dtree* (*tail T*) = *tail T*
⟨*proof*⟩

lemma *dhead-aux-elem-eq-head*:
t = *to-dtree-aux r* ⟹ *e* ∈ *darcs t* ⟹ *dhead t def e* = *head T e*
⟨*proof*⟩

lemma *dhead-elem-eq-head*: *e* ∈ *darcs to-dtree* ⟹ *dhead to-dtree def e* = *head T e*
⟨*proof*⟩

lemma *to-dtree-dhead-eq-head-aux*: *dhead to-dtree* (*head T*) *e* = *head T e*
⟨*proof*⟩

lemma *to-dtree-dhead-eq-head*: *dhead to-dtree* (*head T*) = *head T*
⟨*proof*⟩

lemma *from-to-dtree-eq-orig*: *from-dtree* (*to-dtree*) = *T*
⟨*proof*⟩

lemma *subtree-darc-tail-parent*:
⟦*is-subtree* (*Node r xs*) *to-dtree*; (*t*,*e*) ∈ *fset xs*⟧ ⟹ *tail T e* = *r*
⟨*proof*⟩

lemma *subtree-darc-head-root*:
⟦*is-subtree* (*Node r xs*) *to-dtree*; (*t*,*e*) ∈ *fset xs*⟧ ⟹ *head T e* = *Dtree.root t*
⟨*proof*⟩

lemma *subtree-darc-in-arcs*:
⟦*is-subtree* (*Node r xs*) *to-dtree*; (*t*,*e*) ∈ *fset xs*⟧ ⟹ *e* ∈ *arcs T*
⟨*proof*⟩

lemma *subtree-child-dom*: ⟦*is-subtree* (*Node r xs*) *to-dtree*; (*t*,*e*) ∈ *fset xs*⟧ ⟹ *r* →$_T$ *Dtree.root t*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wf-dtree-sub*: *is-subtree x t* ⟹ *wf-dtree x*
⟨*proof*⟩

lemma *root-not-subtree*: ⟦(*Node r xs*) = *t*; *x* ∈ *fst* ‘ *fset xs*⟧ ⟹ *r* ∉ *dverts x*
⟨*proof*⟩

lemma *dverts-child-subset*: ⟦(*Node r xs*) = *t*; *x* ∈ *fst* ‘ *fset xs*⟧ ⟹ *dverts x* ⊂ *dverts t*
⟨*proof*⟩

lemma *child-arc-not-subtree*: ⟦(*Node r xs*) = *t*; (*x*,*e1*) ∈ *fset xs*⟧ ⟹ *e1* ∉ *darcs x*
⟨*proof*⟩

lemma *darcs-child-subset*: ⟦(*Node r xs*) = *t*; *x* ∈ *fst* ‘ *fset xs*⟧ ⟹ *darcs x* ⊂ *darcs t*
⟨*proof*⟩

lemma *dtail-in-dverts*: *e* ∈ *darcs t* ⟹ *dtail t def e* ∈ *dverts t*
⟨*proof*⟩

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*⟩

lemma *dhead-in-dverts*: *e* ∈ *darcs t* ⟹ *dhead t def e* ∈ *dverts t*
⟨*proof*⟩

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*⟩

lemma *dhead-in-dverts-no-root*: *e* ∈ *darcs t* ⟹ *dhead t def e* ∈ (*dverts t* − {*root t*})
⟨*proof*⟩

lemma *dhead-in-childverts-no-root*:
assumes *e* ∈ *darcs x* **and** (*x*,*e*′) ∈ *fset xs* **and** *Node r xs* = *t*
shows *dhead t def e* ∈ (*dverts x* − {*root x*})
⟨*proof*⟩

lemma *dtree-cas-iff-subtree*:
assumes (*x*,*e1*) ∈ *fset xs* **and** *Node r xs* = *t* **and** *set p* ⊆ *darcs x*
shows *pre-digraph.cas* (*from-dtree dt dh x*) *u p v*
⟷ *pre-digraph.cas* (*from-dtree dt dh t*) *u p v*

(**is** *pre-digraph.cas ?X - - - ⟷ pre-digraph.cas ?T - - -*)
⟨*proof*⟩

lemma *dtree-cas-exists*:
v ∈ dverts t ⟹ ∃ p. set p ⊆ darcs t ∧ pre-digraph.cas (from-dtree dt dh t) (root t) p v
⟨*proof*⟩

lemma *dtree-awalk-exists*:
assumes *v ∈ dverts t*
shows *∃ p. pre-digraph.awalk (from-dtree dt dh t) (root t) p v*
⟨*proof*⟩

lemma *subtree-root-not-root*: *t = Node r xs ⟹ (x,e) ∈ fset xs ⟹ root x ≠ r*
⟨*proof*⟩

lemma *dhead-not-root*:
assumes *e ∈ darcs t*
shows *dhead t def e ≠ root t*
⟨*proof*⟩

lemma *nohead-cas-no-arc-in-subset*:
⟦∀ e∈darcs t. dhead t dh e ≠ v; p≠[]; pre-digraph.cas (from-dtree dt dh t) u p v⟧
⟹ ¬set p ⊆ darcs t
⟨*proof*⟩

lemma *dtail-root-in-set*:
assumes *e ∈ darcs t* **and** *t = Node r xs* **and** *dtail t dt e = r*
shows *e ∈ snd ‘ fset xs*
⟨*proof*⟩

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*⟩

lemma *subtree-uneq-if-arc-uneq*:
⟦(x1,e1) ∈ fset xs; (x2,e2) ∈ fset xs; e1≠e2; Node r xs = t⟧ ⟹ x1 ≠ x2
⟨*proof*⟩

lemma *arc-uneq-if-subtree-uneq*:
⟦(x1,e1) ∈ fset xs; (x2,e2) ∈ fset xs; x1≠x2; Node r xs = t⟧ ⟹ e1 ≠ e2
⟨*proof*⟩

lemma *dhead-unique*: *e ∈ darcs t ⟹ p ∈ darcs t ⟹ e ≠ p ⟹ dhead t dh e ≠ dhead t dh p*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟧

$\Longrightarrow ps = es$
⟨*proof*⟩

lemma *dtree-unique-awalk-exists*:
assumes $v \in dverts\ t$
shows $\exists!p.\ pre\text{-}digraph.awalk\ (from\text{-}dtree\ dt\ dh\ t)\ (root\ t)\ p\ v$
⟨*proof*⟩

lemma *from-dtree-directed*: *directed-tree* (*from-dtree dt dh t*) (*root t*)
⟨*proof*⟩

theorem *from-dtree-fin-directed*: *finite-directed-tree* (*from-dtree dt dh t*) (*root t*)
⟨*proof*⟩

7.3.3 Identity of Transformation Operations

lemma *dhead-img-eq-root-img*:
$Node\ r\ xs = t$
$\Longrightarrow (\lambda e.\ ((dhead\ (Node\ r\ xs)\ dh\ e),\ e))\ `\ snd\ `\ fset\ xs = (\lambda(x,e).\ (root\ x,\ e))\ `\ fset\ xs$
⟨*proof*⟩

lemma *childarcs-in-out-arcs*:
$[\![Node\ r\ xs = t;\ e \in snd\ `\ fset\ xs]\!] \Longrightarrow e \in out\text{-}arcs\ (from\text{-}dtree\ dt\ dh\ t)\ r$
⟨*proof*⟩

lemma *out-arcs-in-childarcs*:
assumes $Node\ r\ xs = t$ **and** $e \in out\text{-}arcs\ (from\text{-}dtree\ dt\ dh\ t)\ r$
shows $e \in snd\ `\ fset\ xs$
⟨*proof*⟩

lemma *childarcs-eq-out-arcs*:
$Node\ r\ xs = t \Longrightarrow snd\ `\ fset\ xs = out\text{-}arcs\ (from\text{-}dtree\ dt\ dh\ t)\ r$
⟨*proof*⟩

lemma *dtail-in-subtree-eq-subtree*:
$[\![is\text{-}subtree\ t1\ t;\ e \in darcs\ t1]\!] \Longrightarrow dtail\ t\ def\ e = dtail\ t1\ def\ e$
⟨*proof*⟩

lemma *dtail-in-subdverts*:
assumes $e \in darcs\ x$ **and** $is\text{-}subtree\ x\ t$
shows $dtail\ t\ def\ e \in dverts\ x$
⟨*proof*⟩

lemma *dhead-in-subtree-eq-subtree*:
$[\![is\text{-}subtree\ t1\ t;\ e \in darcs\ t1]\!] \Longrightarrow dhead\ t\ def\ e = dhead\ t1\ def\ e$
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *subarcs-eq-out-arcs*:
is-subtree (*Node r xs*) *t* ⟹ *snd* ‘ *fset xs* = *out-arcs* (*from-dtree dt dh t*) *r*
⟨*proof*⟩

lemma *dhead-sub-img-eq-root-img*:
is-subtree (*Node v ys*) *t*
⟹ (λ*e*. ((*dhead t dh e*), *e*)) ‘ *snd* ‘ *fset ys* = (λ(*x*,*e*). (*root x*, *e*)) ‘ *fset ys*
⟨*proof*⟩

lemma *subtree-to-dtree-aux-eq*:
assumes *is-subtree x t* **and** *v* ∈ *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*
∧ *finite-directed-tree.to-dtree-aux* (*from-dtree dt dh x*) (*root x*) = *x*
⟨*proof*⟩

interpretation *T*: *finite-directed-tree from-dtree dt dh t root t*
⟨*proof*⟩

lemma *to-from-dtree-aux-id*: *T.to-dtree-aux dt dh* (*root t*) = *t*
⟨*proof*⟩

theorem *to-from-dtree-id*: *T.to-dtree dt dh* = *t*
⟨*proof*⟩

end

context *finite-directed-tree*
begin

lemma *wf-to-dtree-aux*: *wf-dtree* (*to-dtree-aux r*)
⟨*proof*⟩

theorem *wf-to-dtree*: *wf-dtree to-dtree*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mdeg0-if-fcard0*: *fcard xs* = *0* ⟹ *max-deg* (*Node r xs*) = *0*
⟨*proof*⟩

lemma *mdeg0-iff-fcard0*: *fcard xs* = *0* ⟷ *max-deg* (*Node r xs*) = *0*
⟨*proof*⟩

lemma *nempty-if-mdeg-gt-fcard*: *max-deg* (*Node r xs*) > *fcard xs* ⟹ *xs* ≠ {||}
⟨*proof*⟩

lemma *mdeg-img-nempty*: *max-deg* (*Node r xs*) > *fcard xs* ⟹ *max-deg* ‘ *fst* ‘ *fset xs* ≠ {}
⟨*proof*⟩

lemma *mdeg-img-fin*: *finite* (*max-deg* ‘ *fst* ‘ *fset xs*)
⟨*proof*⟩

lemma *mdeg-Max-if-gt-fcard*:
max-deg (*Node r xs*) > *fcard xs* ⟹ *max-deg* (*Node r xs*) = *Max* (*max-deg* ‘ *fst* ‘ *fset xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *maxif-eq-Max*: *finite X* $\Longrightarrow$ (*if X* $\neq$ {} *then max x* (*Max X*) *else x*) = *Max* (*insert x X*)
⟨*proof*⟩

lemma *mdeg-img-empty-iff*: *max-deg* ‘ *fst* ‘ *fset xs* = {} $\longleftrightarrow$ *xs* = {||}
⟨*proof*⟩

lemma *mdeg-alt*: *max-deg* (*Node r xs*) = *Max* (*insert* (*fcard xs*) (*max-deg* ‘ *fst* ‘ *fset xs*))
⟨*proof*⟩

lemma *finite-fMax-union*: *finite Y* $\Longrightarrow$ *finite* ($\bigcup$ *y*$\in$*Y*. {*Max* (*f y*)})
⟨*proof*⟩

lemma *Max-union-Max-out*:
assumes *finite Y* **and** $\forall$ *y* $\in$ *Y*. *finite* (*f y*) **and** $\forall$ *y* $\in$ *Y*. *f y* $\neq$ {} **and** *Y* $\neq$ {}
shows *Max* ($\bigcup$ *y*$\in$*Y*. {*Max* (*f y*)}) = *Max* ($\bigcup$ *y*$\in$*Y*. *f y*) (**is** *?M1*=-)
⟨*proof*⟩

lemma *Max-union-Max-out-insert*:
⟦*finite Y*; $\forall$ *y* $\in$ *Y*. *finite* (*f y*); $\forall$ *y* $\in$ *Y*. *f y* $\neq$ {}; *Y* $\neq$ {}⟧
$\Longrightarrow$ *Max* (*insert x* ($\bigcup$ *y*$\in$*Y*. {*Max* (*f y*)})) = *Max* (*insert x* ($\bigcup$ *y*$\in$*Y*. *f y*))
⟨*proof*⟩

lemma *mdeg-alt2*: *max-deg t* = *Max* {*fcard* (*sucs x*)|*x*. *is-subtree x t*}
⟨*proof*⟩

lemma *mdeg-singleton*: *max-deg* (*Node r* {|(*t1*,*e1*)|}) = *max* (*max-deg t1*) (*fcard* {|(*t1*,*e1*)|})
⟨*proof*⟩

lemma *mdeg-ge-child-aux*: (*t1*,*e1*) $\in$ *fset xs* $\Longrightarrow$ *max-deg t1* $\leq$ *Max* (*max-deg* ‘ *fst* ‘ *fset xs*)
⟨*proof*⟩

lemma *mdeg-ge-child*: (*t1*,*e1*) $\in$ *fset xs* $\Longrightarrow$ *max-deg t1* $\leq$ *max-deg* (*Node r xs*)
⟨*proof*⟩

lemma *mdeg-ge-child′*: *t1* $\in$ *fst* ‘ *fset xs* $\Longrightarrow$ *max-deg t1* $\leq$ *max-deg* (*Node r xs*)
⟨*proof*⟩

lemma *mdeg-ge-sub*: *is-subtree t1 t2* $\Longrightarrow$ *max-deg t1* $\leq$ *max-deg t2*
⟨*proof*⟩

lemma *mdeg-gt-0-if-nempty*: *xs* $\neq$ {||} $\Longrightarrow$ *max-deg* (*Node r xs*) > *0*
⟨*proof*⟩

corollary *empty-if-mdeg-0*: *max-deg* (*Node r xs*) = *0* $\Longrightarrow$ *xs* = {||}
⟨*proof*⟩

lemma *nempty-if-mdeg-n0*: *max-deg* (*Node r xs*) ≠ *0* ⟹ *xs* ≠ {||}
⟨*proof*⟩

corollary *empty-iff-mdeg-0*: *max-deg* (*Node r xs*) = *0* ⟷ *xs* = {||}
⟨*proof*⟩

lemma *mdeg-root*: *max-deg* (*Node r xs*) = *max-deg* (*Node v xs*)
⟨*proof*⟩

lemma *mdeg-ge-fcard*: *fcard xs* ≤ *max-deg* (*Node r xs*)
⟨*proof*⟩

lemma *mdeg-fcard-if-fcard-ge-child*:
∀ (*t*,*e*) ∈ *fset xs*. *max-deg t* ≤ *fcard xs* ⟹ *max-deg* (*Node r xs*) = *fcard xs*
⟨*proof*⟩

lemma *mdeg-fcard-if-fcard-ge-child′*:
∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *fcard xs* ⟹ *max-deg* (*Node r xs*) = *fcard xs*
⟨*proof*⟩

lemma *fcard-single-1*: *fcard* {|*x*|} = *1*
⟨*proof*⟩

lemma *fcard-single-1-iff*: *fcard xs* = *1* ⟷ (∃ *x*. *xs* = {|*x*|})
⟨*proof*⟩

lemma *fcard-not0-if-elem*: ∃ *x*. *x* ∈ *fset xs* ⟹ *fcard xs* ≠ *0*
⟨*proof*⟩

lemma *fcard1-if-le1-elem*: ⟦*fcard xs* ≤ *1*; *x* ∈ *fset xs*⟧ ⟹ *fcard xs* = *1*
⟨*proof*⟩

lemma *singleton-if-fcard-le1-elem*: ⟦*fcard xs* ≤ *1*; *x* ∈ *fset xs*⟧ ⟹ *xs* = {|*x*|}
⟨*proof*⟩

lemma *singleton-if-mdeg-le1-elem*: ⟦*max-deg* (*Node r xs*) ≤ *1*; *x* ∈ *fset xs*⟧ ⟹ *xs* = {|*x*|}
⟨*proof*⟩

lemma *singleton-if-mdeg-le1-elem-suc*: ⟦*max-deg t* ≤ *1*; *x* ∈ *fset* (*sucs t*)⟧ ⟹ *sucs t* = {|*x*|}
⟨*proof*⟩

lemma *fcard0-if-le1-not-singleton*: ⟦∀ *x*. *xs* ≠ {|*x*|}; *fcard xs* ≤ *1*⟧ ⟹ *fcard xs* = *0*
⟨*proof*⟩

lemma *empty-fset-if-fcard-le1-not-singleton*: ⟦∀ *x*. *xs* ≠ {|*x*|}; *fcard xs* ≤ *1*⟧ ⟹ *xs*

$= \{||\}$
⟨*proof*⟩

lemma *fcard0-if-mdeg-le1-not-single*: ⟦∀ *x*. *xs* ≠ {|*x*|}; *max-deg* (*Node r xs*) ≤ *1*⟧ ⟹ *fcard xs* = *0*
⟨*proof*⟩

lemma *empty-fset-if-mdeg-le1-not-single*: ⟦∀ *x*. *xs* ≠ {|*x*|}; *max-deg* (*Node r xs*) ≤ *1*⟧ ⟹ *xs* = {||}
⟨*proof*⟩

lemma *fcard0-if-mdeg-le1-not-single-suc*:
⟦∀ *x*. *sucs t* ≠ {|*x*|}; *max-deg t* ≤ *1*⟧ ⟹ *fcard* (*sucs t*) = *0*
⟨*proof*⟩

lemma *empty-fset-if-mdeg-le1-not-single-suc*: ⟦∀ *x*. *sucs t* ≠ {|*x*|}; *max-deg t* ≤ *1*⟧ ⟹ *sucs t* = {||}
⟨*proof*⟩

lemma *mdeg-1-singleton*:
assumes *max-deg* (*Node r xs*) = *1*
shows ∃ *x*. *xs* = {|*x*|}
⟨*proof*⟩

lemma *subtree-child-if-dvert-notr-mdeg-le1*:
assumes *max-deg* (*Node r xs*) ≤ *1* **and** *v* ≠ *r* **and** *v* ∈ *dverts* (*Node r xs*)
shows ∃ *r′ e zs*. *is-subtree* (*Node r′* {|(*Node v zs*,*e*)|}) (*Node r xs*)
⟨*proof*⟩

lemma *subtree-child-if-dvert-notroot-mdeg-le1*:
⟦*max-deg t* ≤ *1*; *v* ≠ *root t*; *v* ∈ *dverts t*⟧
⟹ ∃ *r′ e zs*. *is-subtree* (*Node r′* {|(*Node v zs*,*e*)|}) *t*
⟨*proof*⟩

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*⟩

lemma *mdeg-child-sucs-le*: *max-deg* (*Node v* (*sucs t*)) ≤ *max-deg* (*Node r* {|(*t*,*e*)|})
⟨*proof*⟩

lemma *mdeg-eq-child-if-singleton-gt1*:
max-deg (*Node r* {|(*t1*,*e1*)|}) > *1* ⟹ *max-deg* (*Node r* {|(*t1*,*e1*)|}) = *max-deg t1*
⟨*proof*⟩

lemma *fcard-gt1-if-mdeg-gt-child*:
assumes *max-deg* (*Node r xs*) > *n* **and** *t1* ∈ *fst* ‘ *fset xs* **and** *max-deg t1* ≤ *n*

and *n≠0*
shows *fcard xs > 1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fcard-gt1-if-mdeg-gt-child1*:
⟦*max-deg* (*Node r xs*) *> 1*; *t1 ∈ fst ‘ fset xs*; *max-deg t1 ≤ 1*⟧ ⟹ *fcard xs > 1*
⟨*proof*⟩

lemma *fcard-gt1-if-mdeg-gt-suc1*:
⟦*max-deg t2 > 1*; *t1 ∈ fst ‘ fset* (*sucs t2*); *max-deg t1 ≤ 1*⟧ ⟹ *fcard* (*sucs t2*) *> 1*
⟨*proof*⟩

lemma *fcard-lt-non-inj-f*:
⟦*f a = f b*; *a ∈ fset xs*; *b ∈ fset xs*; *a≠b*⟧ ⟹ *fcard* (*f |‘| xs*) *< fcard xs*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mdeg-img-eq*:

assumes $\forall (t,e) \in fset\ xs.\ max\text{-}deg\ (fst\ (f\ (t,e))) = max\text{-}deg\ t$
and $fcard\ (f\ |`|\ xs) = fcard\ xs$
shows $max\text{-}deg\ (Node\ r\ (f\ |`|\ xs)) = max\text{-}deg\ (Node\ r\ xs)$
⟨*proof*⟩

lemma *num-leaves-1-if-mdeg-1*: $max\text{-}deg\ t \leq 1 \Longrightarrow num\text{-}leaves\ t = 1$
⟨*proof*⟩

lemma *num-leaves-ge1*: $num\text{-}leaves\ t \geq 1$
⟨*proof*⟩

lemma *num-leaves-ge-card*: $num\text{-}leaves\ (Node\ r\ xs) \geq fcard\ xs$
⟨*proof*⟩

lemma *num-leaves-root*: $num\text{-}leaves\ (Node\ r\ xs) = num\text{-}leaves\ (Node\ r'\ xs)$
⟨*proof*⟩

lemma *num-leaves-singleton*: $num\text{-}leaves\ (Node\ r\ \{|(t,e)|\}) = num\text{-}leaves\ t$
⟨*proof*⟩

7.5 List Conversions

function $dtree\text{-}to\text{-}list :: ('a,'b)\ dtree \Rightarrow ('a \times 'b)\ list$ **where**
$dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t,e)|\}) = (root\ t,e)\ \#\ dtree\text{-}to\text{-}list\ t$
$| \forall x.\ xs \neq \{|x|\} \Longrightarrow dtree\text{-}to\text{-}list\ (Node\ r\ xs) = []$
⟨*proof*⟩
termination ⟨*proof*⟩

fun $dtree\text{-}from\text{-}list :: 'a \Rightarrow ('a \times 'b)\ list \Rightarrow ('a,'b)\ dtree$ **where**
$dtree\text{-}from\text{-}list\ r\ [] = Node\ r\ \{||\}$
$| dtree\text{-}from\text{-}list\ r\ ((v,e)\#xs) = Node\ r\ \{|(dtree\text{-}from\text{-}list\ v\ xs,\ e)|\}$

fun $wf\text{-}list\text{-}arcs :: ('a \times 'b)\ list \Rightarrow bool$ **where**
$wf\text{-}list\text{-}arcs\ [] = True$
$| wf\text{-}list\text{-}arcs\ ((v,e)\#xs) = (e \notin snd\ `\ set\ xs \wedge wf\text{-}list\text{-}arcs\ xs)$

fun $wf\text{-}list\text{-}verts :: ('a \times 'b)\ list \Rightarrow bool$ **where**
$wf\text{-}list\text{-}verts\ [] = True$
$| wf\text{-}list\text{-}verts\ ((v,e)\#xs) = (v \notin fst\ `\ set\ xs \wedge wf\text{-}list\text{-}verts\ xs)$

lemma *dtree-to-list-sub-dverts-ins*:
$insert\ (root\ t)\ (fst\ `\ set\ (dtree\text{-}to\text{-}list\ t)) \subseteq dverts\ t$
⟨*proof*⟩

lemma *dtree-to-list-eq-dverts-ins*:
$max\text{-}deg\ t \leq 1 \Longrightarrow insert\ (root\ t)\ (fst\ `\ set\ (dtree\text{-}to\text{-}list\ t)) = dverts\ t$
⟨*proof*⟩

lemma *dtree-to-list-eq-dverts-sucs*:

max-deg t $\leq$ *1* $\Longrightarrow$ *fst* ‘ *set* (*dtree-to-list t*) = ($\bigcup$ *x* $\in$ *fset* (*sucs t*). *dverts* (*fst x*))
⟨*proof*⟩

lemma *dtree-to-list-sub-dverts*:
wf-dverts t $\Longrightarrow$ *fst* ‘ *set* (*dtree-to-list t*) $\subseteq$ *dverts t* − {*root t*}
⟨*proof*⟩

lemma *dtree-to-list-eq-dverts*:
⟦*wf-dverts t*; *max-deg t* $\leq$ *1*⟧ $\Longrightarrow$ *fst* ‘ *set* (*dtree-to-list t*) = *dverts t* − {*root t*}
⟨*proof*⟩

lemma *dtree-to-list-eq-dverts-single*:
⟦*max-deg t* $\leq$ *1*; *sucs t* = {|(*t1*,*e1*)|}⟧ $\Longrightarrow$ *fst* ‘ *set* (*dtree-to-list t*) = *dverts t1*
⟨*proof*⟩

lemma *dtree-to-list-sub-darcs*: *snd* ‘ *set* (*dtree-to-list t*) $\subseteq$ *darcs t*
⟨*proof*⟩

lemma *dtree-to-list-eq-darcs*: *max-deg t* $\leq$ *1* $\Longrightarrow$ *snd* ‘ *set* (*dtree-to-list t*) = *darcs t*
⟨*proof*⟩

lemma *dtree-from-list-eq-dverts*: *dverts* (*dtree-from-list r xs*) = *insert r* (*fst* ‘ *set xs*)
⟨*proof*⟩

lemma *dtree-from-list-eq-darcs*: *darcs* (*dtree-from-list r xs*) = *snd* ‘ *set xs*
⟨*proof*⟩

lemma *dtree-from-list-root-r*[*simp*]: *root* (*dtree-from-list r xs*) = *r*
⟨*proof*⟩

lemma *dtree-from-list-v-eq-r*:
Node r xs = *dtree-from-list v ys* $\Longrightarrow$ *r* = *v*
⟨*proof*⟩

lemma *dtree-from-list-fcard0-empty*: *fcard* (*sucs* (*dtree-from-list r* [])) = *0*
⟨*proof*⟩

lemma *dtree-from-list-fcard0-iff-empty*: *fcard* (*sucs* (*dtree-from-list r xs*)) = *0* $\longleftrightarrow$ *xs* = []
⟨*proof*⟩

lemma *dtree-from-list-fcard1-iff-nempty*: *fcard* (*sucs* (*dtree-from-list r xs*)) = *1* $\longleftrightarrow$ *xs* $\neq$ []
⟨*proof*⟩

lemma *dtree-from-list-fcard-le1*: *fcard* (*sucs* (*dtree-from-list r xs*)) $\leq$ *1*
⟨*proof*⟩

lemma *dtree-from-empty-deg-0*: *max-deg* (*dtree-from-list r* []) = *0*
⟨*proof*⟩

lemma *dtree-from-list-deg-le-1*: *max-deg* (*dtree-from-list r xs*) ≤ *1*
⟨*proof*⟩

lemma *dtree-from-list-deg-1*: *xs* ≠ [] ⟷ *max-deg* (*dtree-from-list r xs*) = *1*
⟨*proof*⟩

lemma *dtree-from-list-singleton*: *xs* ≠ [] ⟹ ∃ *t e*. *dtree-from-list r xs* = *Node r* {|(*t*,*e*)|}
⟨*proof*⟩

lemma *dtree-from-to-list-id*: *max-deg t* ≤ *1* ⟹ *dtree-from-list* (*root t*) (*dtree-to-list t*) = *t*
⟨*proof*⟩

lemma *dtree-to-from-list-id*: *dtree-to-list* (*dtree-from-list r xs*) = *xs*
⟨*proof*⟩

lemma *dtree-from-list-eq-singleton-hd*:
Node r0 {|(*t0*,*e0*)|} = *dtree-from-list v1 ys* ⟹ (∃ *xs*. (*root t0*, *e0*) # *xs* = *ys*)
⟨*proof*⟩

lemma *dtree-from-list-eq-singleton*:
Node r0 {|(*t0*,*e0*)|} = *dtree-from-list v1 ys* ⟹ *r0* = *v1* ∧ (∃ *xs*. (*root t0*, *e0*) # *xs* = *ys*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *dtree-from-list-eq-empty*:
Node r {||} = *dtree-from-list v ys* ⟹ *r* = *v* ∧ *ys* = []
⟨*proof*⟩

lemma *dtree-from-list-sucs-cases*:
Node r xs = *dtree-from-list v ys* ⟹ *xs* = {||} ∨ (∃ *x*. *xs* = {|*x*|})
⟨*proof*⟩

lemma *dtree-from-list-uneq-sequence-xs*:
strict-subtree (*Node r0 xs0*) (*dtree-from-list v1 ys*)
$\Longrightarrow \exists\, e\ as\ bs.\ as\ @\ (r0,e)\ \#\ bs = ys \wedge Node\ r0\ xs0 = dtree\text{-}from\text{-}list\ r0\ bs$
⟨*proof*⟩

lemma *dtree-from-list-sequence-xs*:
⟦*is-subtree* (*Node r xs*) (*dtree-from-list v1 ys*)⟧
$\Longrightarrow \exists\, e\ as\ bs.\ as\ @\ (r,e)\ \#\ bs = ((v1,e1)\#ys) \wedge Node\ r\ xs = dtree\text{-}from\text{-}list\ r\ bs$
⟨*proof*⟩

lemma *dtree-from-list-sequence-dverts*:
⟦*is-subtree* (*Node r xs*) (*dtree-from-list v1 ys*)⟧
$\Longrightarrow \exists\, e\ as\ bs.\ as\ @\ (r,e)\ \#\ bs = ((v1,e1)\#ys) \wedge dverts\ (Node\ r\ xs) = insert\ r\ (fst\ `\ set\ bs)$
⟨*proof*⟩

lemma *dtree-from-list-dverts-subset-set*:
$set\ bs \subseteq set\ ds \Longrightarrow dverts\ (dtree\text{-}from\text{-}list\ r\ bs) \subseteq dverts\ (dtree\text{-}from\text{-}list\ r\ ds)$
⟨*proof*⟩

lemma *wf-darcs'-iff-wf-list-arcs*: $wf\text{-}list\text{-}arcs\ xs \longleftrightarrow wf\text{-}darcs'\ (dtree\text{-}from\text{-}list\ r\ xs)$
⟨*proof*⟩

lemma *wf-darcs-iff-wf-list-arcs*: $wf\text{-}list\text{-}arcs\ xs \longleftrightarrow wf\text{-}darcs\ (dtree\text{-}from\text{-}list\ r\ xs)$
⟨*proof*⟩

lemma *wf-dverts-iff-wf-list-verts*:
$r \notin fst\ `\ set\ xs \wedge wf\text{-}list\text{-}verts\ xs \longleftrightarrow wf\text{-}dverts\ (dtree\text{-}from\text{-}list\ r\ xs)$
⟨*proof*⟩

theorem *wf-dtree-iff-wf-list*:
$wf\text{-}list\text{-}arcs\ xs \wedge r \notin fst\ `\ set\ xs \wedge wf\text{-}list\text{-}verts\ xs \longleftrightarrow wf\text{-}dtree\ (dtree\text{-}from\text{-}list\ r\ xs)$
⟨*proof*⟩

lemma *wf-list-arcs-if-wf-darcs*: $wf\text{-}darcs\ t \Longrightarrow wf\text{-}list\text{-}arcs\ (dtree\text{-}to\text{-}list\ t)$
⟨*proof*⟩

lemma *wf-list-verts-if-wf-dverts*: $wf\text{-}dverts\ t \Longrightarrow wf\text{-}list\text{-}verts\ (dtree\text{-}to\text{-}list\ t)$
⟨*proof*⟩

lemma *distinct-if-wf-list-arcs*: $wf\text{-}list\text{-}arcs\ xs \Longrightarrow distinct\ xs$
⟨*proof*⟩

lemma *distinct-if-wf-list-verts*: $wf\text{-}list\text{-}verts\ xs \Longrightarrow distinct\ xs$
⟨*proof*⟩

lemma *wf-list-arcs-alt*: $wf\text{-}list\text{-}arcs\ xs \longleftrightarrow distinct\ (map\ snd\ xs)$

⟨*proof*⟩

lemma *wf-list-verts-alt*: *wf-list-verts xs* ⟷ *distinct* (*map fst xs*)
⟨*proof*⟩

lemma *subtree-from-list-split-eq-if-wfverts*:
assumes *wf-list-verts* (*as*@(*r*,*e*)#*bs*)
and *v* ∉ *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*⟩

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*))⟧
⟹ *Node r xs* = *dtree-from-list r bs*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dtree-to-list-subtree-dverts-eq-fsts*:
⟦*max-deg t* ≤ *1*; *strict-subtree* (*Node r xs*) *t*⟧

$\Longrightarrow \exists as\ e\ bs.\ dtree\text{-}to\text{-}list\ t = as@(r,e)\#bs \wedge insert\ r\ (fst\ `\ set\ bs) = dverts\ (Node\ r\ xs)$
⟨*proof*⟩

lemma *dtree-to-list-subtree-dverts-eq-fsts'*:
⟦*max-deg t ≤ 1*; *strict-subtree* (*Node r xs*) *t*⟧
$\Longrightarrow \exists as\ e\ bs.\ dtree\text{-}to\text{-}list\ t = as@(r,e)\#bs \wedge (fst\ `\ set\ ((r,e)\#bs)) = dverts\ (Node\ r\ xs)$
⟨*proof*⟩

lemma *dtree-to-list-split-subtree*:
assumes $as@(r,e)\#bs = dtree\text{-}to\text{-}list\ t$
shows $\exists xs.\ strict\text{-}subtree\ (Node\ r\ xs)\ t \wedge dtree\text{-}to\text{-}list\ (Node\ r\ xs) = bs$
⟨*proof*⟩

lemma *dtree-to-list-split-subtree-dverts-eq-fsts*:
assumes *max-deg t ≤ 1* **and** $as@(r,e)\#bs = dtree\text{-}to\text{-}list\ t$
shows $\exists xs.\ strict\text{-}subtree\ (Node\ r\ xs)\ t \wedge dverts\ (Node\ r\ xs) = insert\ r\ (fst`set\ bs)$
⟨*proof*⟩

lemma *dtree-to-list-split-subtree-dverts-eq-fsts'*:
assumes *max-deg t ≤ 1* **and** $as@(r,e)\#bs = dtree\text{-}to\text{-}list\ t$
shows $\exists xs.\ strict\text{-}subtree\ (Node\ r\ xs)\ t \wedge dverts\ (Node\ r\ xs) = (fst\ `\ set\ ((r,e)\#bs))$
⟨*proof*⟩

lemma *dtree-from-list-dverts-subset-wfdverts1*:
assumes $dverts\ t1 \subseteq fst\ `\ set\ ((r,e2)\#bs)$
and $wf\text{-}dverts\ (dtree\text{-}from\text{-}list\ v\ (as@(r,e2)\#bs))$
and $is\text{-}subtree\ (Node\ r\ ys)\ (dtree\text{-}from\text{-}list\ v\ (as@(r,e2)\#bs))$
shows $dverts\ t1 \subseteq dverts\ (Node\ r\ ys)$
⟨*proof*⟩

lemma *dtree-from-list-dverts-subset-wfdverts1'*:
assumes $wf\text{-}dverts\ (dtree\text{-}from\text{-}list\ v\ cs)$
and $is\text{-}subtree\ (Node\ r\ ys)\ (dtree\text{-}from\text{-}list\ v\ cs)$
and $\exists as\ e\ bs.\ as@(r,e)\#bs = cs \wedge dverts\ t1 \subseteq fst\ `\ set\ ((r,e)\#bs)$
shows $dverts\ t1 \subseteq dverts\ (Node\ r\ ys)$
⟨*proof*⟩

lemma *dtree-from-list-1-leaf*: $num\text{-}leaves\ (dtree\text{-}from\text{-}list\ r\ xs) = 1$
⟨*proof*⟩

7.6 Inserting in Dtrees

abbreviation *insert-before* ::
$'a \Rightarrow 'b \Rightarrow 'a \Rightarrow (('a,'b)\ dtree \times 'b)\ fset \Rightarrow (('a,'b)\ dtree \times 'b)\ fset$ **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* :: $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'a \Rightarrow ('a,'b)$ *dtree* $\Rightarrow ('a,'b)$ *dtree* **where**
insert-between v e x y (*Node r xs*) = (*if* $x{=}r \wedge (\exists\, t.\ t \in$ *fst* ‘ *fset xs* $\wedge$ *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*: $x \notin$ *dverts t* $\Longrightarrow$ *insert-between v e x y t* = *t*
⟨*proof*⟩

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*⟩

lemma *insert-before-commute*:
comp-fun-commute (λ(*t1*,*e1*). *finsert* (*if root t1* = *y1 then* (*Node v* $\{|$(*t1*,*e1*)$|\}$,*e*) *else* (*t1*,*e1*)))
⟨*proof*⟩

interpretation *Comm*:
comp-fun-commute λ(*t1*,*e1*). *finsert* (*if root t1* = *y then* (*Node v* $\{|$(*t1*,*e1*)$|\}$,*e*) *else* (*t1*,*e1*))
⟨*proof*⟩

lemma *root-not-new-in-orig*:
⟦(*t1*,*e1*) ∈ *fset* (*insert-before v e y xs*); *root t1* ≠ *v*⟧ $\Longrightarrow$ (*t1*,*e1*) ∈ *fset xs*
⟨*proof*⟩

lemma *root-not-y-in-new*:
⟦(*t1*,*e1*) ∈ *fset xs*; *root t1* ≠ *y*⟧ $\Longrightarrow$ (*t1*,*e1*) ∈ *fset* (*insert-before v e y xs*)
⟨*proof*⟩

lemma *root-noty-if-in-insert-before*:
⟦(*t1*,*e1*) ∈ *fset* (*insert-before v e y xs*); *v*≠*y*⟧ $\Longrightarrow$ *root t1* ≠ *y*
⟨*proof*⟩

lemma *in-insert-before-child-in-orig*:
⟦(*t1*,*e1*) ∈ *fset* (*insert-before v e y xs*); (*t1*,*e1*) ∉ *fset xs*⟧
$\Longrightarrow \exists$ (*t2*,*e2*) ∈ *fset xs*. (*Node v* $\{|$(*t2*,*e2*)$|\}$) = *t1* ∧ *root t2* = *y* ∧ *e1*=*e*
⟨*proof*⟩

lemma *insert-before-not-y-id*:
¬(∃ *t*. *t* ∈ *fst* ‘ *fset xs* ∧ *root t* = *y*) $\Longrightarrow$ *insert-before v e y xs* = *xs*
⟨*proof*⟩

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,e)) |‘| xs
⟨*proof*⟩

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*⟩

lemma *insert-between-add-v-if-x-in*:
x ∈ dverts t ⟹ dverts (insert-between v e x y t) = insert v (dverts t)
⟨*proof*⟩

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*⟩

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) ∈ fset (insert-before v e y xs)*
and *(t2,e2) ∈ fset (insert-before v e y xs)*
and *(t1,e1) ≠ (t2,e2)*
shows *dverts t1 ∩ dverts t2 = {}*
⟨*proof*⟩

lemma *disjoint-dverts-aux1′*:
assumes *wf-dverts (Node r xs)* **and** *v ∉ dverts (Node r xs)*
shows *∀ (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)
⟨*proof*⟩

lemma *insert-before-wf-dverts*:
⟦∀ (t,e1) ∈ fset xs. wf-dverts t; v ∉ dverts(Node r xs); (t1,e1) ∈ fset (insert-before v e y xs)⟧
⟹ wf-dverts t1
⟨*proof*⟩

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⟩

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*)
⟨*proof*⟩

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*⟩

lemma *insert-between-wf-dverts*: *v* ∉ *dverts t* ⟹ *wf-dverts* (*insert-between v e x y t*)
⟨*proof*⟩

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*) ∪ *Basic-BNFs.snds x*)
= *insert e* (⋃ *x*∈*fset xs*. ⋃ (*darcs* ‘ *Basic-BNFs.fsts x*) ∪ *Basic-BNFs.snds x*)
⟨*proof*⟩

lemma *insert-between-add-e-if-x-in*:
x ∈ *dverts t* ⟹ *darcs* (*insert-between v e x y t*) = *insert e* (*darcs t*)
⟨*proof*⟩

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*⟩

lemma *disjoint-darcs-aux1-aux2*:
assumes *disjoint-darcs xs*
and $e \notin$ *darcs* (*Node r xs*)
and (*t1*,*e1*) $\in$ *fset* (*insert-before v e y xs*)
shows *e1* $\notin$ *darcs t1*
⟨*proof*⟩

lemma *disjoint-darcs-aux1*:
assumes *wf-dverts* (*Node r xs*) **and** $v \notin$ *dverts* (*Node r xs*)
and *wf-darcs* (*Node r xs*) **and** $e \notin$ *darcs* (*Node r xs*)
shows *disjoint-darcs* (*insert-before v e y xs*) (**is** *disjoint-darcs ?xs*)
⟨*proof*⟩

lemma *insert-before-wf-darcs*:
⟦*wf-darcs* (*Node r xs*); $e \notin$ *darcs* (*Node r xs*); (*t1*,*e1*) $\in$ *fset* (*insert-before v e y xs*)⟧
$\Longrightarrow$ *wf-darcs t1*
⟨*proof*⟩

lemma *disjoint-darcs-aux2*:
assumes *wf-darcs* (*Node r xs*) **and** $e \notin$ *darcs* (*Node r xs*)
shows *disjoint-darcs* (*finsert* (*Node v* {||},*e*) *xs*)
⟨*proof*⟩

lemma *disjoint-darcs-aux3-aux1*:
assumes (*t*,*e1*) $\in$ *fset xs*
and $x \in$ *dverts t*
and *wf-darcs* (*Node r xs*)
and $e \notin$ *darcs* (*Node r xs*)
and (*t2*,*e2*) $\in$ (λ(*t1*,*e1*). (*insert-between v e x y t1*, *e1*)) ‘ *fset xs*
and (*t3*,*e3*) $\in$ (λ(*t1*,*e1*). (*insert-between v e x y t1*, *e1*)) ‘ *fset xs*
and (*t2*,*e2*)$\neq$(*t3*,*e3*)
and *wf-dverts* (*Node r xs*)
shows (*darcs t2* $\cup$ {*e2*}) $\cap$ (*darcs t3* $\cup$ {*e3*}) = {}
⟨*proof*⟩

lemma *disjoint-darcs-aux3-aux2*:
assumes (*t*,*e1*) $\in$ *fset xs*
and $x \in$ *dverts t*
and *wf-darcs* (*Node r xs*)
and $e \notin$ *darcs* (*Node r xs*)
and (*t2*,*e2*) $\in$ (λ(*t1*,*e1*). (*insert-between v e x y t1*, *e1*)) ‘ *fset xs*
and *wf-dverts* (*Node r xs*)
shows *e2* $\notin$ *darcs t2*
⟨*proof*⟩

lemma *disjoint-darcs-aux3*:
assumes (*t*,*e1*) $\in$ *fset xs*

and $x \in$ *dverts t*
and *wf-darcs* (*Node r xs*)
and $e \notin$ *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*⟩

lemma *insert-between-wf-darcs*:
⟦$e \notin$ *darcs t*; $v \notin$ *dverts t* ⟧ ⟹ *wf-darcs* (*insert-between v e x y t*)
⟨*proof*⟩

theorem *insert-between-wf-dtree*:
⟦$e \notin$ *darcs t*; $v \notin$ *dverts t* ⟧ ⟹ *wf-dtree* (*insert-between v e x y t*)
⟨*proof*⟩

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*⟩

lemma *snds-neq-img-snds-neq*:
assumes ∀(*t*,*e*) ∈ *fset xs*. ∀(*t2*,*e2*) ∈ *fset xs*. *e*≠*e2* ∨ (*t*,*e*) = (*t2*,*e2*)
shows ∀(*t1*,*e1*) ∈ *fset* ((λ(*t1*,*e1*). (*f t1*, *e1*)) |‘| *xs*).
∀(*t2*,*e2*) ∈ *fset* ((λ(*t1*,*e1*). (*f t1*, *e1*)) |‘| *xs*). *e1*≠*e2* ∨ (*t1*,*e1*) = (*t2*,*e2*)
⟨*proof*⟩

lemma *snds-neq-if-disjoint-darcs*:
assumes *disjoint-darcs xs*
shows ∀(*t*,*e*) ∈ *fset xs*. ∀(*t2*,*e2*) ∈ *fset xs*. *e*≠*e2* ∨ (*t*,*e*) = (*t2*,*e2*)
⟨*proof*⟩

lemma *snds-neq-img-card-eq*:
assumes ∀(*t*,*e*) ∈ *fset xs*. ∀(*t2*,*e2*) ∈ *fset xs*. *e*≠*e2* ∨ (*t*,*e*) = (*t2*,*e2*)
shows *fcard* ((λ(*t1*,*e1*). (*f t1*, *e1*)) |‘| *xs*) = *fcard xs*
⟨*proof*⟩

lemma *fst-neq-img-card-eq*:
assumes ∀(*t*,*e*) ∈ *fset xs*. ∀(*t2*,*e2*) ∈ *fset xs*. *f t* ≠ *f t2* ∨ (*t*,*e*) = (*t2*,*e2*)
shows *fcard* ((λ(*t1*,*e1*). (*f t1*, *e1*)) |‘| *xs*) = *fcard xs*
⟨*proof*⟩

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*⟩

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*) =

$(v \neq [] \wedge (\forall\, v2 \in fst\ `\ set\ xs.\ set\ v \cap set\ v2 = \{\}) \wedge wf\text{-}list\text{-}lverts\ xs)$

8.2 List Dtrees as Well-Formed Dtrees

lemma *list-in-verts-if-lverts*: $x \in dlverts\ t \Longrightarrow (\exists\, v \in dverts\ t.\ x \in set\ v)$
⟨*proof*⟩

lemma *list-in-verts-iff-lverts*: $x \in dlverts\ t \longleftrightarrow (\exists\, v \in dverts\ t.\ x \in set\ v)$
⟨*proof*⟩

lemma *lverts-if-in-verts*: $[\![v \in dverts\ t;\ x \in set\ v]\!] \Longrightarrow x \in dlverts\ t$
⟨*proof*⟩

lemma *nempty-inter-notin-dverts*: $[\![v \neq [];\ set\ v \cap dlverts\ t = \{\}]\!] \Longrightarrow v \notin dverts\ t$
⟨*proof*⟩

lemma *empty-notin-wf-dlverts*: $wf\text{-}dlverts\ t \Longrightarrow [] \notin dverts\ t$
⟨*proof*⟩

lemma *wf-dlverts'-rec*: $[\![wf\text{-}dlverts'\ (Node\ r\ xs);\ t1 \in fst\ `\ fset\ xs]\!] \Longrightarrow wf\text{-}dlverts'\ t1$
⟨*proof*⟩

lemma *wf-dlverts'-suc*: $[\![wf\text{-}dlverts'\ t;\ t1 \in fst\ `\ fset\ (sucs\ t)]\!] \Longrightarrow wf\text{-}dlverts'\ t1$
⟨*proof*⟩

lemma *wf-dlverts-suc*: $[\![wf\text{-}dlverts\ t;\ t1 \in fst\ `\ fset\ (sucs\ t)]\!] \Longrightarrow wf\text{-}dlverts\ t1$
⟨*proof*⟩

lemma *wf-dlverts-subtree*: $[\![wf\text{-}dlverts\ t;\ is\text{-}subtree\ t1\ t]\!] \Longrightarrow wf\text{-}dlverts\ t1$
⟨*proof*⟩

lemma *dlverts-eq-dverts-union*: $dlverts\ t = \bigcup\ (set\ `\ dverts\ t)$
⟨*proof*⟩

lemma *dlverts-eq-dverts-union'*: $dlverts\ t = (\bigcup x \in dverts\ t.\ set\ x)$
⟨*proof*⟩

lemma *dverts-nempty*: $dverts\ t \neq \{\}$
⟨*proof*⟩

lemma *dlverts-nempty-aux*: $[] \notin dverts\ t \Longrightarrow dlverts\ t \neq \{\}$
⟨*proof*⟩

lemma *dlverts-nempty-if-wf*: $wf\text{-}dlverts\ t \Longrightarrow dlverts\ t \neq \{\}$
⟨*proof*⟩

lemma *nempty-root-in-lverts*: $root\ t \neq [] \Longrightarrow hd\ (root\ t) \in dlverts\ t$

⟨*proof*⟩

lemma *roothd-in-lverts-if-wf*: *wf-dlverts t* ⟹ *hd* (*root t*) ∈ *dlverts t*
⟨*proof*⟩

lemma *hd-in-lverts-if-wf*: ⟦*wf-dlverts t*; *v* ∈ *dverts t*⟧ ⟹ *hd v* ∈ *dlverts t*
⟨*proof*⟩

lemma *dlverts-notin-root-sucs*:
⟦*wf-dlverts t*; *t1* ∈ *fst* ‘ *fset* (*sucs t*); *x* ∈ *dlverts t1*⟧ ⟹ *x* ∉ *set* (*root t*)
⟨*proof*⟩

lemma *dverts-inter-empty-if-verts-inter*:
assumes *dlverts x* ∩ *dlverts y* = {} **and** *wf-dlverts x*
shows *dverts x* ∩ *dverts y* = {}
⟨*proof*⟩

lemma *disjoint-dlverts-if-wf*: *wf-dlverts t* ⟹ *disjoint-dlverts* (*sucs t*)
⟨*proof*⟩

lemma *disjoint-dlverts-subset*:
assumes *xs* |⊆| *ys* **and** *disjoint-dlverts ys*
shows *disjoint-dlverts xs*
⟨*proof*⟩

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* = {}
⟨*proof*⟩

lemma *wf-dlverts-sub*:
assumes *xs* |⊆| *ys* **and** *wf-dlverts* (*Node r ys*)
shows *wf-dlverts* (*Node r xs*)
⟨*proof*⟩

lemma *wf-dlverts-sucs*: ⟦*wf-dlverts t*; *x* ∈ *fset* (*sucs t*)⟧ ⟹ *wf-dlverts* (*Node* (*root t*) {|*x*|})
⟨*proof*⟩

lemma *wf-dverts-if-wf-dlverts*: *wf-dlverts t* ⟹ *wf-dverts t*
⟨*proof*⟩

lemma *notin-dlverts-child-if-wf-in-root*:
⟦*wf-dlverts* (*Node r xs*); *x* ∈ *set r*; *t* ∈ *fst* ‘ *fset xs*⟧ ⟹ *x* ∉ *dlverts t*
⟨*proof*⟩

lemma *notin-dlverts-suc-if-wf-in-root*:
⟦*wf-dlverts t1*; *x* ∈ *set* (*root t1*); *t2* ∈ *fst* ‘ *fset* (*sucs t1*)⟧ ⟹ *x* ∉ *dlverts t2*
⟨*proof*⟩

lemma *root-if-same-lvert-wf*:
⟦*wf-dlverts* (*Node r xs*); *x* ∈ *set r*; *v* ∈ *dverts* (*Node r xs*); *x* ∈ *set v*⟧ ⟹ *v* = *r*
⟨*proof*⟩

lemma *dverts-same-if-set-wf*:
⟦*wf-dlverts t*; *v1* ∈ *dverts t*; *v2* ∈ *dverts t*; *x* ∈ *set v1*; *x* ∈ *set v2*⟧ ⟹ *v1* = *v2*
⟨*proof*⟩

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*⟩

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*⟩

lemma *vert-disjoint-if-not-root*:
assumes *wf-dlverts t*
and *v* ∈ *dverts t* − {*root t*}
shows *set* (*root t*) ∩ *set v* = {}
⟨*proof*⟩

lemma *vert-disjoint-if-to-list*:
⟦*wf-dlverts* (*Node r* {|(*t1*,*e1*)|}); *v* ∈ *fst* ‘ *set* (*dtree-to-list t1*)⟧
⟹ *set* (*root t1*) ∩ *set v* = {}
⟨*proof*⟩

lemma *wf-list-lverts-if-wf-dlverts*: *wf-dlverts t* ⟹ *wf-list-lverts* (*dtree-to-list t*)
⟨*proof*⟩

lemma *child-in-dlverts*: (*t1*,*e*) ∈ *fset xs* ⟹ *dlverts t1* ⊆ *dlverts* (*Node r xs*)
⟨*proof*⟩

lemma *suc-in-dlverts*: (*t1*,*e*) ∈ *fset* (*sucs t2*) ⟹ *dlverts t1* ⊆ *dlverts t2*
⟨*proof*⟩

lemma *suc-in-dlverts′*: *t1* ∈ *fst* ‘ *fset* (*sucs t2*) ⟹ *dlverts t1* ⊆ *dlverts t2*
⟨*proof*⟩

lemma *subtree-in-dlverts*: *is-subtree t1 t2* ⟹ *dlverts t1* ⊆ *dlverts t2*
⟨*proof*⟩

lemma *subtree-root-if-dlverts*: *x* ∈ *dlverts t* ⟹ ∃ *r xs*. *is-subtree* (*Node r xs*) *t* ∧ *x* ∈ *set r*
⟨*proof*⟩

lemma *x-not-root-strict-subtree*:
assumes $x \in$ *dlverts t* **and** $x \notin$ *set* (*root t*)
shows $\exists$ *r xs t1*. *is-subtree* (*Node r xs*) *t* $\wedge$ *t1* $\in$ *fst* ' *fset xs* $\wedge$ $x \in$ *set* (*root t1*)
⟨*proof*⟩

lemma *dverts-disj-if-wf-dlverts*:
⟦*wf-dlverts t*; *v1* $\in$ *dverts t*; *v2* $\in$ *dverts t*; *v1* $\neq$ *v2*⟧ $\Longrightarrow$ *set v1* $\cap$ *set v2* = {}
⟨*proof*⟩

thm *empty-notin-wf-dlverts*

lemma *wf-dlverts'-if-dlverts*: *wf-dlverts t* $\Longrightarrow$ *wf-dlverts' t*
⟨*proof*⟩

lemma *disjoint-dlverts-if-wf'-aux*:
assumes *wf-dlverts'* (*Node r xs*)
and (*t1*,*e1*) $\in$ *fset xs*
and (*t2*,*e2*) $\in$ *fset xs*
and (*t1*,*e1*) $\neq$ (*t2*,*e2*)
shows *dlverts t1* $\cap$ *dlverts t2* = {}
⟨*proof*⟩

lemma *disjoint-dlverts-if-wf'*: *wf-dlverts'* (*Node r xs*) $\Longrightarrow$ *disjoint-dlverts xs*
⟨*proof*⟩

lemma *root-nempty-if-wf'*: *wf-dlverts'* (*Node r xs*) $\Longrightarrow$ *r* $\neq$ []
⟨*proof*⟩

lemma *disjoint-root-if-wf'-aux*:
assumes *wf-dlverts'* (*Node r xs*)
and (*t1*,*e1*) $\in$ *fset xs*
shows *set r* $\cap$ *dlverts t1* = {}
⟨*proof*⟩

lemma *disjoint-root-if-wf'*:
wf-dlverts' (*Node r xs*) $\Longrightarrow$ $\forall$ (*t1*,*e1*) $\in$ *fset xs*. *set r* $\cap$ *dlverts t1* = {}
⟨*proof*⟩

lemma *wf-dlverts-if-dlverts'*: *wf-dlverts' t* $\Longrightarrow$ *wf-dlverts t*
⟨*proof*⟩

lemma *wf-dlverts-iff-dlverts'*: *wf-dlverts t* $\longleftrightarrow$ *wf-dlverts' t*
⟨*proof*⟩

locale *list-dtree* =
fixes *t* :: ('*a list*,'*b*) *dtree*
assumes *wf-arcs*: *wf-darcs t*
and *wf-lverts*: *wf-dlverts t*

sublocale *list-dtree* $\subseteq$ *wf-dtree*
⟨*proof*⟩

theorem *list-dtree-iff-wf-list*:
wf-list-arcs xs $\wedge$ $(\forall\, v \in$ *fst* ‘ *set xs*. *set r* $\cap$ *set v* $= \{\})$ $\wedge$ $r \neq []$ $\wedge$ *wf-list-lverts xs*
$\longleftrightarrow$ *list-dtree* (*dtree-from-list r xs*)
⟨*proof*⟩

lemma *list-dtree-subset*:
assumes *xs* $|\subseteq|$ *ys* **and** *list-dtree* (*Node r ys*)
shows *list-dtree* (*Node r xs*)
⟨*proof*⟩

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*) $\in$ *fset xs* **and** (*y*,*e2*) $\in$ *fset xs* **and** (*x*,*e1*)$\neq$(*y*,*e2*)
shows *dlverts x* $\cap$ *dlverts y* $= \{\}$
⟨*proof*⟩

lemma *wf-dlverts-to-dtree-aux*: $[\![r \in$ *verts T*; *t* = *to-dtree-aux r*$]\!] \Longrightarrow$ *wf-dlverts t*
⟨*proof*⟩

lemma *wf-dlverts-to-dtree*: *wf-dlverts to-dtree*
⟨*proof*⟩

theorem *list-dtree-to-dtree*: *list-dtree to-dtree*
⟨*proof*⟩

end

context *list-dtree*
begin

lemma *list-dtree-rec*: $[\![$*Node r xs* = *t*; (*x*,*e*) $\in$ *fset xs*$]\!] \Longrightarrow$ *list-dtree x*
⟨*proof*⟩

lemma *list-dtree-rec-suc*: (*x*,*e*) $\in$ *fset* (*sucs t*) $\Longrightarrow$ *list-dtree x*
⟨*proof*⟩

lemma *list-dtree-sub*: *is-subtree x t* $\Longrightarrow$ *list-dtree x*
⟨*proof*⟩

theorem *from-dtree-fin-list-dir*: *fin-list-directed-tree* (*root t*) (*from-dtree dt dh t*)
⟨*proof*⟩

8.3 Combining Preserves Well-Formedness

lemma *remove-child-sub*: *remove-child x xs* $|\subseteq|$ *xs*
⟨*proof*⟩

lemma *child2-commute-aux*:
assumes $f = (\lambda(t,\text{-})\ b.$ *case t of Node r ys* $\Rightarrow$ *if r = a then ys* $|\cup|$ *b else b*)
shows $(f\ y \circ f\ x)\ z = (f\ x \circ f\ y)\ z$
⟨*proof*⟩

lemma *child2-commute*:
comp-fun-commute $(\lambda(t,\text{-})\ b.$ *case t of Node r ys* $\Rightarrow$ *if r = x then ys* $|\cup|$ *b else b*)
⟨*proof*⟩

interpretation *Comm*:
comp-fun-commute $\lambda(t,\text{-})\ b.$ *case t of Node r ys* $\Rightarrow$ *if r = x then ys* $|\cup|$ *b else b*
⟨*proof*⟩

lemma *input-in-child2*:
zs $|\subseteq|$ *child2 x zs ys*
⟨*proof*⟩

lemma *child2-subset-if-input1*:
zs' $|\subseteq|$ *zs* $\Longrightarrow$ *child2 x zs' ys* $|\subseteq|$ *child2 x zs ys*
⟨*proof*⟩

lemma *child2-subset-if-input2*:
ys' $|\subseteq|$ *ys* $\Longrightarrow$ *child2 x xs ys'* $|\subseteq|$ *child2 x xs ys*
⟨*proof*⟩

lemma *darcs-split*: *darcs* (*Node r* (*xs*$|\cup|$*ys*)) = *darcs* (*Node r xs*) $\cup$ *darcs* (*Node r ys*)
⟨*proof*⟩

lemma *darcs-sub-if-children-sub*: *xs* $|\subseteq|$ *ys* $\Longrightarrow$ *darcs* (*Node r xs*) $\subseteq$ *darcs* (*Node v ys*)
⟨*proof*⟩

lemma *darc-in-child2-snd-if-nin-fst*:
e $\in$ *darcs* (*Node x* (*child2 a xs ys*)) $\Longrightarrow$ *e* $\notin$ *darcs* (*Node v ys*) $\Longrightarrow$ *e* $\in$ *darcs* (*Node r xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *darcs-child2-sub*: *darcs* (*Node x* (*child2 y xs ys*)) $\subseteq$ *darcs* (*Node r xs*) $\cup$ *darcs* (*Node r' ys*)

⟨*proof*⟩

lemma *darcs-combine-sub-orig*: *darcs* (*combine x y t1*) ⊆ *darcs t1*
⟨*proof*⟩

lemma *child2-in-child*:
⟦*b* ∈ *fset* (*child2 a ys xs*); *b* |∉| *ys*⟧ ⟹ ∃ *rs e*. (*Node a rs*, *e*) ∈ *fset xs* ∧ *b* |∈| *rs*
⟨*proof*⟩

lemma *child-in-darcs*: (*y*,*e2*) ∈ *fset xs* ⟹ *darcs y* ∪ {*e2*} ⊆ *darcs* (*Node r xs*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *disjoint-darcs-combine*:
assumes *Node r xs* = *t*
shows *disjoint-darcs* ((λ(*t*,*e*). (*combine x y t*,*e*)) |`| *xs*)
⟨*proof*⟩

lemma *wf-darcs-combine*: *wf-darcs* (*combine x y t*)
⟨*proof*⟩

lemma *v-in-dlverts-if-in-comb*: *v* ∈ *dlverts* (*combine x y t*) ⟹ *v* ∈ *dlverts t*
⟨*proof*⟩

lemma *ex-subtree-if-in-lverts*: *v* ∈ *dlverts t1* ⟹ ∃ *t2*. *is-subtree t2 t1* ∧ *v* ∈ *set* (*root t2*)
⟨*proof*⟩

lemma *child'-in-child2*:
assumes (*Node y rs1*,*e1*) ∈ *fset xs* **and** (*t2*,*e2*) ∈ *fset rs1*
shows (*t2*,*e2*) ∈ *fset* (*child2 y ys xs*)
⟨*proof*⟩

lemma *v-in-comb-if-in-dlverts*: *v* ∈ *dlverts t* ⟹ *v* ∈ *dlverts* (*combine x y t*)
⟨*proof*⟩

lemma *dlverts-comb-id*[*simp*]: *dlverts* (*combine x y t*) = *dlverts t*
⟨*proof*⟩

lemma *wf-dlverts-comb-aux*:

assumes $\forall (t,e) \in fset\ xs.\ dlverts\ (combine\ x\ y\ t) = dlverts\ t$
and $\forall (t1,e1) \in fset\ xs.\ \forall (t2,e2) \in fset\ xs.\ dlverts\ t1 \cap dlverts\ t2 = \{\} \vee (t1,e1)=(t2,e2)$
and $(t1,e1) \in fset\ ((\lambda(t,e).\ (combine\ x\ y\ t,\ e))\ |`|\ xs)$
and $(t2,e2) \in fset\ ((\lambda(t,e).\ (combine\ x\ y\ t,\ e))\ |`|\ xs)$
shows $dlverts\ t1 \cap dlverts\ t2 = \{\} \vee (t1,e1)=(t2,e2)$
⟨*proof*⟩

lemma *wf-dlverts-child2*:
assumes $(t1,e) \in fset\ (child2\ y\ (remove\text{-}child\ y\ xs)\ xs)$
and $\forall (t,e) \in fset\ xs.\ wf\text{-}dlverts\ t$
shows *wf-dlverts t1*
⟨*proof*⟩

lemma *wf-dlverts-child2-aux1*:
assumes $(t1,e1) \in fset\ (child2\ y\ (remove\text{-}child\ y\ xs)\ xs)$
and $\exists t.\ t \in fst\ `\ fset\ xs \wedge root\ t = y$
and *wf-dlverts* $(Node\ r\ xs)$
shows $set\ (r@y) \cap dlverts\ t1 = \{\}$
⟨*proof*⟩

lemma *wf-dlverts-child2-aux2*:
assumes $\forall (t1,e1) \in fset\ xs.\ \forall (t2,e2) \in fset\ xs.\ dlverts\ t1 \cap dlverts\ t2 = \{\} \vee (t1,e1)=(t2,e2)$
and $\forall (t,e) \in fset\ xs.\ wf\text{-}dlverts\ t$
and $(t1,e1) \in fset\ (child2\ y\ (remove\text{-}child\ y\ xs)\ xs)$
and $(t2,e2) \in fset\ (child2\ y\ (remove\text{-}child\ y\ xs)\ xs)$
and $(t1,e1) \neq (t2,e2)$
shows $dlverts\ t1 \cap dlverts\ t2 = \{\}$
⟨*proof*⟩

lemma *wf-dlverts-combine*: *wf-dlverts* $(combine\ x\ y\ t)$
⟨*proof*⟩

theorem *list-dtree-comb*: *list-dtree* $(combine\ x\ y\ t)$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *set-merge*: *set xs ∪ set ys = set* (*Sorting-Algorithms.merge cmp xs ys*)
⟨*proof*⟩

lemma *input-empty-if-merge-empty*: *Sorting-Algorithms.merge cmp xs ys = [] ⟹ xs = [] ∧ ys = []*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *wf-list-lverts-merge*:
⟦*wf-list-lverts xs*; *wf-list-lverts ys*;
$\forall v1 \in fst\ `\ set\ xs.\ \forall v2 \in fst\ `\ set\ ys.\ set\ v1 \cap set\ v2 = \{\}$⟧
⟹ wf-list-lverts (*Sorting-Algorithms.merge cmp xs ys*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *merge-split-supset-fst*:

assumes *as@(r,e)#bs = (Sorting-Algorithms.merge cmp xs ys)*
shows *∃ as′ bs′. set bs′ ⊆ set bs ∧ (as′@(r,e)#bs′ = xs ∨ as′@(r,e)#bs′ = ys)*
⟨*proof*⟩

lemma *merge-split-supset′*:
assumes *r ∈ set (Sorting-Algorithms.merge cmp xs ys)*
shows *∃ as bs as′ bs′. as@r#bs = (Sorting-Algorithms.merge cmp xs ys)*
∧ set bs′ ⊆ set bs ∧ (as′@r#bs′ = xs ∨ as′@r#bs′ = ys)
⟨*proof*⟩

lemma *merge-split-supset-fst′*:
assumes *r ∈ fst ‘ set (Sorting-Algorithms.merge cmp xs ys)*
shows *∃ as e bs as′ bs′. as@(r,e)#bs = (Sorting-Algorithms.merge cmp xs ys)*
∧ set bs′ ⊆ set bs ∧ (as′@(r,e)#bs′ = xs ∨ as′@(r,e)#bs′ = ys)
⟨*proof*⟩

lemma *merge-split-supset-subtree*:
assumes *∀ as bs. as@(r,e)#bs = xs ⟶*
(∃ zs. is-subtree (Node r zs) t ∧ dverts (Node r zs) ⊆ fst ‘ set ((r,e)#bs))
and *∀ as bs. as@(r,e)#bs = ys ⟶*
(∃ zs. is-subtree (Node r zs) t ∧ dverts (Node r zs) ⊆ fst ‘ set ((r,e)#bs))
and *as@(r,e)#bs = (Sorting-Algorithms.merge cmp xs ys)*
shows *∃ zs. is-subtree (Node r zs) t ∧ dverts (Node r zs) ⊆ (fst ‘ set ((r,e)#bs))*
⟨*proof*⟩

lemma *merge-split-supset-strict-subtree*:
assumes *∀ as bs. as@(r,e)#bs = xs ⟶ (∃ zs. strict-subtree (Node r zs) t*
∧ dverts (Node r zs) ⊆ fst ‘ set ((r,e)#bs))
and *∀ as bs. as@(r,e)#bs = ys ⟶ (∃ zs. strict-subtree (Node r zs) t*
∧ dverts (Node r zs) ⊆ fst ‘ set ((r,e)#bs))
and *as@(r,e)#bs = (Sorting-Algorithms.merge cmp xs ys)*
shows *∃ zs. strict-subtree (Node r zs) t*
∧ dverts (Node r zs) ⊆ (fst ‘ set ((r,e)#bs))
⟨*proof*⟩

lemma *sorted-app-l*: *sorted cmp (xs@ys) ⟹ sorted cmp xs*
⟨*proof*⟩

lemma *sorted-app-r*: *sorted cmp (xs@ys) ⟹ sorted cmp ys*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ranked-dtree-rec-suc*: (*x*,*e*) ∈ *fset* (*sucs t*) ⟹ *ranked-dtree x cmp*
⟨*proof*⟩

lemma *ranked-dtree-subtree*: *is-subtree x t* ⟹ *ranked-dtree x cmp*
⟨*proof*⟩

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)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dtree-to-list-x-in-dverts*:
x ∈ *fst* ‘ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})) ⟹ *x* ∈ *dverts t1*
⟨*proof*⟩

lemma *dtree-to-list-x-in-dlverts*:
x ∈ *fst* ‘ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})) ⟹ *set x* ⊆ *dlverts t1*
⟨*proof*⟩

lemma *dtree-to-list-x1-disjoint*:
dlverts t1 ∩ *dlverts t2* = {}

$\Longrightarrow \forall\, x1 \in fst\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\}))\ .\ set\ x1 \cap dlverts\ t2 = \{\}$

⟨*proof*⟩

lemma *dtree-to-list-xs-disjoint*:
$dlverts\ t1 \cap dlverts\ t2 = \{\}$
$\Longrightarrow \forall\, x1 \in fst\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\}))\ .$
$\forall\, x2 \in fst\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r'\ \{|(t2,e2)|\}))\ .\ set\ x1 \cap set\ x2 = \{\}$

⟨*proof*⟩

lemma *dtree-to-list-e-in-darcs*:
$e \in snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\})) \Longrightarrow e \in darcs\ t1 \cup \{e1\}$

⟨*proof*⟩

lemma *dtree-to-list-e-disjoint*:
$(darcs\ t1 \cup \{e1\}) \cap (darcs\ t2 \cup \{e2\}) = \{\}$
$\Longrightarrow \forall\, e \in snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\}))\ .\ e \notin darcs\ t2 \cup \{e2\}$

⟨*proof*⟩

lemma *dtree-to-list-es-disjoint*:
$(darcs\ t1 \cup \{e1\}) \cap (darcs\ t2 \cup \{e2\}) = \{\}$
$\Longrightarrow \forall\, e3 \in snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\}))\ .$
$\forall\, e4 \in snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r'\ \{|(t2,e2)|\}))\ .\ e3 \neq e4$

⟨*proof*⟩

lemma *dtree-to-list-xs-not-equiv*:
assumes $dlverts\ t1 \cap dlverts\ t2 = \{\}$
and $(darcs\ t1 \cup \{e3\}) \cap (darcs\ t2 \cup \{e4\}) = \{\}$
and $(x1,e1) \in set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e3)|\}))$ **and** $x1 \neq []$
and $(x2,e2) \in set\ (dtree\text{-}to\text{-}list\ (Node\ r'\ \{|(t2,e4)|\}))$ **and** $x2 \neq []$
shows $compare\ cmp'\ (x1,e1)\ (x2,e2) \neq Equiv$

⟨*proof*⟩

lemma *merge-dtree1-not-equiv*:
assumes $dlverts\ t1 \cap dlverts\ t2 = \{\}$
and $(darcs\ t1 \cup \{e1\}) \cap (darcs\ t2 \cup \{e2\}) = \{\}$
and $[] \notin dverts\ t1$
and $[] \notin dverts\ t2$
and $xs = dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\})$
and $ys = dtree\text{-}to\text{-}list\ (Node\ r'\ \{|(t2,e2)|\})$
shows $\forall\, (x1,e1) \in set\ xs.\ \forall\, (x2,e2) \in set\ ys.\ compare\ cmp'\ (x1,e1)\ (x2,e2) \neq Equiv$

⟨*proof*⟩

lemma *merge-commute-aux1*:
assumes $dlverts\ t1 \cap dlverts\ t2 = \{\}$
and $(darcs\ t1 \cup \{e1\}) \cap (darcs\ t2 \cup \{e2\}) = \{\}$
and $[] \notin dverts\ t1$
and $[] \notin dverts\ t2$

and $xs = dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\})$
and $ys = dtree\text{-}to\text{-}list\ (Node\ r'\ \{|(t2,e2)|\})$
shows $Sorting\text{-}Algorithms.merge\ cmp'\ xs\ ys = Sorting\text{-}Algorithms.merge\ cmp'\ ys\ xs$
⟨*proof*⟩

lemma *dtree-to-list-x1-list-disjoint*:
$set\ x2 \cap dlverts\ t1 = \{\}$
$\Longrightarrow \forall\, x1 \in fst\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\}))$. $set\ x1 \cap set\ x2 = \{\}$
⟨*proof*⟩

lemma *dtree-to-list-e1-list-disjoint'*:
$set\ x2 \cap darcs\ t1 \cup \{e1\} = \{\}$
$\Longrightarrow \forall\, x1 \in snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\}))$. $x1 \notin set\ x2$
⟨*proof*⟩

lemma *dtree-to-list-e1-list-disjoint*:
$e2 \notin darcs\ t1 \cup \{e1\}$
$\Longrightarrow \forall\, x1 \in snd\ `\ set\ (dtree\text{-}to\text{-}list\ (Node\ r\ \{|(t1,e1)|\}))$. $x1 \neq e2$
⟨*proof*⟩

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*⟩

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 $Sorting\text{-}Algorithms.merge\ cmp'\ xs\ ys = Sorting\text{-}Algorithms.merge\ cmp'\ ys\ xs$
⟨*proof*⟩

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*⟩

lemma *merge-inter-preserv*:
assumes $f = (merge\text{-}f\ r\ xs)$
and $\neg(\forall\,(v,e) \in set\ z.\ set\ v \cap dlverts\ t1 = \{\} \wedge e \notin darcs\ t1 \cup \{e1\})$
shows $\neg(\forall\,(v,e) \in set\ (f\ (t2,e2)\ z).\ set\ v \cap dlverts\ t1 = \{\} \wedge e \notin darcs\ t1 \cup \{e1\})$
⟨*proof*⟩

lemma *merge-f-eq-z-if-inter'*:
$\neg(\forall\,(v,\text{-}) \in \mathit{set}\ z.\ \mathit{set}\ v \cap \mathit{dlverts}\ t1 = \{\}) \Longrightarrow (\mathit{merge\text{-}f}\ r\ xs)\ (t1,e1)\ z = z$
⟨*proof*⟩

lemma *merge-f-eq-z-if-inter*:
$\neg(\forall\,(v,e) \in \mathit{set}\ z.\ \mathit{set}\ v \cap \mathit{dlverts}\ t1 = \{\} \wedge e \notin \mathit{darcs}\ t1 \cup \{e1\})$
$\Longrightarrow (\mathit{merge\text{-}f}\ r\ xs)\ (t1,e1)\ z = z$
⟨*proof*⟩

lemma *merge-empty-inter-preserv-aux*:
assumes $f = (\mathit{merge\text{-}f}\ r\ xs)$
and $(t2,e2) \in \mathit{fset}\ xs$
and $\forall\,(v,e) \in \mathit{set}\ z.\ \mathit{set}\ v \cap \mathit{dlverts}\ t2 = \{\} \wedge v \neq [] \wedge e \notin \mathit{darcs}\ t2 \cup \{e2\}$
and $\mathit{list\text{-}dtree}\ (\mathit{Node}\ r\ xs)$
and $(t1,e1) \in \mathit{fset}\ xs$
and $(t1,e1) \neq (t2,e2)$
and $\forall\,(v,e) \in \mathit{set}\ z.\ \mathit{set}\ v \cap \mathit{dlverts}\ t1 = \{\} \wedge v \neq [] \wedge e \notin \mathit{darcs}\ t1 \cup \{e1\}$
shows $\forall\,(v,e) \in \mathit{set}\ (f\ (t2,e2)\ z).\ \mathit{set}\ v \cap \mathit{dlverts}\ t1 = \{\} \wedge v \neq [] \wedge e \notin \mathit{darcs}\ t1 \cup \{e1\}$
⟨*proof*⟩

lemma *merge-empty-inter-preserv*:
assumes $f = (\mathit{merge\text{-}f}\ r\ xs)$
and $\forall\,(v,e) \in \mathit{set}\ z.\ \mathit{set}\ v \cap \mathit{dlverts}\ t1 = \{\} \wedge v \neq [] \wedge e \notin \mathit{darcs}\ t1 \cup \{e1\}$
and $(t1,e1) \in \mathit{fset}\ xs$
and $(t1,e1) \neq (t2,e2)$
shows $\forall\,(v,e) \in \mathit{set}\ (f\ (t2,e2)\ z).\ \mathit{set}\ v \cap \mathit{dlverts}\ t1 = \{\} \wedge v \neq [] \wedge e \notin \mathit{darcs}\ t1 \cup \{e1\}$
⟨*proof*⟩

lemma *merge-commute-aux3*:
assumes $f = (\mathit{merge\text{-}f}\ r\ xs)$
and $\mathit{list\text{-}dtree}\ (\mathit{Node}\ r\ xs)$
and $(t1,e1) \neq (t2,e2)$
and $(\forall\,(v,e) \in \mathit{set}\ z.\ \mathit{set}\ v \cap \mathit{dlverts}\ t1 = \{\} \wedge v \neq [] \wedge e \notin \mathit{darcs}\ t1 \cup \{e1\})$
and $(\forall\,(v,e) \in \mathit{set}\ z.\ \mathit{set}\ v \cap \mathit{dlverts}\ t2 = \{\} \wedge v \neq [] \wedge e \notin \mathit{darcs}\ t2 \cup \{e2\})$
and $(t1,e1) \in \mathit{fset}\ xs$
and $(t2,e2) \in \mathit{fset}\ xs$
shows $(f\ (t2,\ e2) \circ f\ (t1,\ e1))\ z = (f\ (t1,\ e1) \circ f\ (t2,\ e2))\ z$
⟨*proof*⟩

lemma *merge-commute-aux*:
assumes $f = (\mathit{merge\text{-}f}\ r\ xs)$
shows $(f\ y \circ f\ x)\ z = (f\ x \circ f\ y)\ z$
⟨*proof*⟩

lemma *merge-commute*: $\mathit{comp\text{-}fun\text{-}commute}\ (\mathit{merge\text{-}f}\ r\ xs)$
⟨*proof*⟩

interpretation *Comm*: *comp-fun-commute merge-f r xs* ⟨*proof*⟩

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*})
⟨*proof*⟩

lemma *disjoint-sets-sucs*: *disjoint-sets* (*sucs t*)
⟨*proof*⟩

lemma *empty-not-elem-subset*:
⟦*xs* |⊆| *ys*; ∀ (*t*,*e*) ∈ *fset ys*. [] ∉ *dverts t*⟧ ⟹ ∀ (*t*,*e*) ∈ *fset xs*. [] ∉ *dverts t*
⟨*proof*⟩

lemma *disjoint-sets-subset*:
assumes *xs* |⊆| *ys* **and** *disjoint-sets ys*
shows *disjoint-sets xs*
⟨*proof*⟩

lemma *merge-mdeg-le-1*: *max-deg* (*merge t1*) ≤ *1*
⟨*proof*⟩

lemma *merge-mdeg-le1-sub*: *is-subtree t1* (*merge t2*) ⟹ *max-deg t1* ≤ *1*
⟨*proof*⟩

lemma *merge-fcard-le1*: *fcard* (*sucs* (*merge t1*)) ≤ *1*
⟨*proof*⟩

lemma *merge-fcard-le1-sub*: *is-subtree t1* (*merge t2*) ⟹ *fcard* (*sucs t1*) ≤ *1*
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ($\lambda a\ b.$ *if* $a \notin$ *fset xs* $\vee \neg$ *Q a b* $\vee \neg$ *P xs then b else R a b*)
⟨*proof*⟩

lemma *merge-ffold-supset*:
assumes *xs* $|\subseteq|$ *ys* **and** *list-dtree* (*Node r ys*)
shows *ffold* (*merge-f r ys*) *acc xs* = *ffold* (*merge-f r xs*) *acc xs*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)|})) []
⟨*proof*⟩

lemma *merge-ffold-empty-inter-preserv*:
⟦*list-dtree* (*Node r ys*); *xs* $|\subseteq|$ *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*⟩

lemma *merge-ffold-empty-inter-preserv'*:
⟦*list-dtree* (*Node r* (*finsert x xs*));
∀(*v*,*e*) ∈ *set z*. *set v* ∩ *dlverts t1* = {} ∧ *v*≠[] ∧ *e* ∉ *darcs t1* ∪ {*e1*};
(*t1*,*e1*) ∈ *fset* (*finsert x xs*); (*t1*,*e1*) ∉ *fset xs*; (*v*,*e*) ∈ *set* (*ffold* (*merge-f r xs*) *z xs*)⟧
⟹ *set v* ∩ *dlverts t1* = {} ∧ *v* ≠ [] ∧ *e* ∉ *darcs t1* ∪ {*e1*}
⟨*proof*⟩

lemma *merge-ffold-set-sub-union*:
list-dtree (*Node r xs*)
⟹ *set* (*ffold* (*merge-f r xs*) [] *xs*) ⊆ ($\bigcup$*x*∈*fset xs*. *set* (*dtree-to-list* (*Node r* {|*x*|})))
⟨*proof*⟩

lemma *merge-ffold-nempty*:
⟦*list-dtree* (*Node r xs*); *xs* ≠ {||}⟧ ⟹ *ffold* (*merge-f r xs*) [] *xs* ≠ []
⟨*proof*⟩

lemma *merge-f-ndisjoint-sets-aux*:
¬*disjoint-sets xs*
⟹ ¬((*t*,*e*) ∈ *fset xs* ∧ *disjoint-sets xs* ∧ (∀ (*v*,-) ∈ *set b*. *set v* ∩ *dlverts t* = {} ∧ *v* ≠ []))
⟨*proof*⟩

lemma *merge-f-not-list-dtree*: ¬*list-dtree* (*Node r xs*) ⟹ (*merge-f r xs*) *a b* = *b*
⟨*proof*⟩

lemma *merge-ffold-empty-if-nwf*: ¬*list-dtree* (*Node r ys*) ⟹ *ffold* (*merge-f r ys*) [] *xs* = []
⟨*proof*⟩

lemma *merge-empty-if-nwf*: ¬*list-dtree* (*Node r xs*) ⟹ *merge* (*Node r xs*) = *Node r* {||}
⟨*proof*⟩

lemma *merge-empty-if-nwf-sucs*: ¬*list-dtree t1* ⟹ *merge t1* = *Node* (*root t1*) {||}
⟨*proof*⟩

lemma *merge-empty*: *merge* (*Node r* {||}) = *Node r* {||}
⟨*proof*⟩

lemma *merge-empty-sucs*:
assumes *sucs t1* = {||}
shows *merge t1* = *Node* (*root t1*) {||}
⟨*proof*⟩

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*)|}
⟨*proof*⟩

lemma *merge-singleton*:
assumes *list-dtree* (*Node r xs*) **and** *xs* ≠ {||}
shows ∃ *t e*. *merge* (*Node r xs*) = *Node r* {|(*t*,*e*)|}
⟨*proof*⟩

lemma *merge-cases*: ∃ *t e*. *merge* (*Node r xs*) = *Node r* {|(*t*,*e*)|} ∨ *merge* (*Node r xs*) = *Node r* {||}
⟨*proof*⟩

lemma *merge-cases-sucs*:
∃ *t e*. *merge t1* = *Node* (*root t1*) {|(*t*,*e*)|} ∨ *merge t1* = *Node* (*root t1*) {||}

⟨*proof*⟩

lemma *merge-single-root*:
(*t2*,*e2*) ∈ *fset* (*sucs* (*merge* (*Node r xs*))) ⟹ *merge* (*Node r xs*) = *Node r* {|(*t2*,*e2*)|}
⟨*proof*⟩

lemma *merge-single-root-sucs*:
(*t2*,*e2*) ∈ *fset* (*sucs* (*merge t1*)) ⟹ *merge t1* = *Node* (*root t1*) {|(*t2*,*e2*)|}
⟨*proof*⟩

lemma *merge-single-root1*:
t2 ∈ *fst* ‘ *fset* (*sucs* (*merge* (*Node r xs*))) ⟹ ∃ *e2*. *merge* (*Node r xs*) = *Node r* {|(*t2*,*e2*)|}
⟨*proof*⟩

lemma *merge-single-root1-sucs*:
t2 ∈ *fst* ‘ *fset* (*sucs* (*merge t1*)) ⟹ ∃ *e2*. *merge t1* = *Node* (*root t1*) {|(*t2*,*e2*)|}
⟨*proof*⟩

lemma *merge-nempty-sucs*: ⟦*list-dtree t1*; *sucs t1* ≠ {||}⟧ ⟹ *sucs* (*merge t1*) ≠ {||}
⟨*proof*⟩

lemma *merge-nempty*: ⟦*list-dtree* (*Node r xs*); *xs* ≠ {||}⟧ ⟹ *sucs* (*merge* (*Node r xs*)) ≠ {||}
⟨*proof*⟩

lemma *merge-xs*: *merge* (*Node r xs*) = *dtree-from-list r* (*ffold* (*merge-f r xs*) [] *xs*)
⟨*proof*⟩

lemma *merge-root-eq*[*simp*]: *root* (*merge t1*) = *root t1*
⟨*proof*⟩

lemma *merge-ffold-fsts-in-childverts*:
⟦*list-dtree* (*Node r xs*); *y* ∈ *fst* ‘ *set* (*ffold* (*merge-f r xs*) [] *xs*)⟧
⟹ ∃ *t1* ∈ *fst* ‘ *fset xs*. *y* ∈ *dverts t1*
⟨*proof*⟩

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*⟩

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*⟩

lemma *merge-ffold-set-eq-union*:
list-dtree (*Node r xs*)
$\Longrightarrow$ *set* (*ffold* (*merge-f r xs*) [] *xs*) = ($\bigcup x \in fset\ xs.$ *set* (*dtree-to-list* (*Node r* {|*x*|})))
⟨*proof*⟩

lemma *sucs-dverts-no-root*:
(*t1*,*e1*) ∈ *fset* (*sucs t*) $\Longrightarrow$ *dverts* (*Node* (*root t*) {|(*t1*,*e1*)|}) − {*root t*} = *dverts t1*
⟨*proof*⟩

lemma *dverts-merge-sub*:
assumes ∀ *t* ∈ *fst* ‘ *fset* (*sucs t0*). *max-deg t* ≤ *1*
shows *dverts* (*merge t0*) ⊆ *dverts t0*
⟨*proof*⟩

lemma *dverts-merge-eq*[*simp*]:
assumes ∀ *t* ∈ *fst* ‘ *fset* (*sucs t*). *max-deg t* ≤ *1*
shows *dverts* (*merge t*) = *dverts t*
⟨*proof*⟩

lemma *dlverts-merge-eq*[*simp*]:
assumes ∀ *t* ∈ *fst* ‘ *fset* (*sucs t*). *max-deg t* ≤ *1*
shows *dlverts* (*merge t*) = *dlverts t*
⟨*proof*⟩

lemma *sucs-darcs-eq-dtree-list*:
assumes (*t1*,*e1*) ∈ *fset* (*sucs t*) **and** *max-deg t1* ≤ *1*
shows *darcs* (*Node* (*root t*) {|(*t1*,*e1*)|}) = *snd* ‘ *set* (*dtree-to-list* (*Node* (*root t*) {|(*t1*,*e1*)|}))
⟨*proof*⟩

lemma *darcs-merge-eq*[*simp*]:
assumes ∀ *t* ∈ *fst* ‘ *fset* (*sucs t*). *max-deg t* ≤ *1*
shows *darcs* (*merge t*) = *darcs t*
⟨*proof*⟩

9.2.4 Merging Preserves Well-Formedness

lemma *dtree-to-list-x-in-darcs*:
x ∈ *snd* ‘ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})) $\Longrightarrow$ *x* ∈ (*darcs t1* ∪ {*e1*})
⟨*proof*⟩

lemma *dtree-to-list-snds-disjoint*:
(*darcs t1* ∪ {*e1*}) ∩ (*darcs t2* ∪ {*e2*}) = {}
$\Longrightarrow$ *snd* ‘ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|})) ∩ (*darcs t2* ∪ {*e2*}) = {}
⟨*proof*⟩

lemma *dtree-to-list-snds-disjoint2*:
(*darcs t1* ∪ {*e1*}) ∩ (*darcs t2* ∪ {*e2*}) = {}
⟹ *snd* ‘ *set* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|}))
∩ *snd* ‘ *set* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|})) = {}
⟨*proof*⟩

lemma *merge-ffold-arc-inter-preserv*:
⟦*list-dtree* (*Node r ys*); *xs* |⊆| *ys*; (*darcs t1* ∪ {*e1*}) ∩ (*snd* ‘ *set z*) = {};
(*t1*,*e1*) ∈ *fset ys*; (*t1*,*e1*) ∉ *fset xs*⟧
⟹ (*darcs t1* ∪ {*e1*}) ∩ (*snd* ‘ *set* (*ffold* (*merge-f r xs*) *z xs*)) = {}
⟨*proof*⟩

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*⟩

lemma *merge-wf-darcs*: *wf-darcs* (*merge t*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *merge-wf-dlverts*: *wf-dlverts* (*merge t*)
⟨*proof*⟩

theorem *merge-list-dtree*: *list-dtree* (*merge t*)
⟨*proof*⟩

corollary *merge-ranked-dtree*: *ranked-dtree* (*merge t*) *cmp*
⟨*proof*⟩

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*⟩

lemma *distinct-merge*:

assumes $\forall v{\in}dverts\ t.\ distinct\ v$ **and** $v{\in}dverts\ (merge\ t)$
shows *distinct v*
⟨*proof*⟩

lemma *merge-hd-root-eq*[*simp*]: *hd* (*root* (*merge t1*)) = *hd* (*root t1*)
⟨*proof*⟩

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*⟩

lemma *merge-ffold-nempty-if-child*:
assumes (*t1*,*e1*) ∈ *fset* (*sucs* (*merge t0*))
shows *ffold* (*merge-f* (*root t0*) (*sucs t0*)) [] (*sucs t0*) ≠ []
⟨*proof*⟩

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*⟩

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*⟩

lemma *ffold-singleton*: *comp-fun-commute f* ⟹ *ffold f z* {|*x*|} = *f x z*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)|}))

⟨*proof*⟩

lemma *merge-disjoint-if-child*:
merge (*Node r* {|(*t1*,*e1*)|}) = *Node r* {|(*t2*,*e2*)|} ⟹ *list-dtree* (*Node r* {|(*t1*,*e1*)|})
⟨*proof*⟩

lemma *merge-root-child-eq*:
merge (*Node r* {|(*t1*,*e1*)|}) = *Node r* {|(*t2*,*e2*)|} ⟹ *root t1* = *root t2*
⟨*proof*⟩

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*⟩

lemma *merge-strict-subtree-dverts-sup*:
assumes ∀ *t* ∈ *fst* ‘ *fset* (*sucs t*). *max-deg t* ≤ *1*
and *strict-subtree* (*Node r xs*) (*merge t*)
shows ∃ *ys*. *is-subtree* (*Node r ys*) *t* ∧ *dverts* (*Node r ys*) ⊆ *dverts* (*Node r xs*)
⟨*proof*⟩

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*⟩

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*⟩

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*)

⟨*proof*⟩
termination ⟨*proof*⟩

lemma *normalize1-size-decr*[*termination-simp*]:
normalize1 t1 ≠ *t1* ⟹ *size* (*normalize1 t1*) < *size t1*
⟨*proof*⟩

lemma *normalize1-size-le*: *size* (*normalize1 t1*) ≤ *size t1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *root-normalize1-eq1'*:
¬*rank* (*rev* (*root t1*)) ≤ *rank* (*rev r*) ⟹ *root* (*normalize1* (*Node r* {|(*t1*,*e1*)|})) = *r*
⟨*proof*⟩

lemma *root-normalize1-eq2*: ∀ *x*. *xs* ≠ {|*x*|} ⟹ *root* (*normalize1* (*Node r xs*)) = *r*
⟨*proof*⟩

lemma *fset-img-eq*: ∀ *x* ∈ *fset xs*. *f x* = *x* ⟹ *f* |‘| *xs* = *xs*
⟨*proof*⟩

lemma *fset-img-uneq*: *f* |‘| *xs* ≠ *xs* ⟹ ∃ *x* ∈ *fset xs*. *f x* ≠ *x*
⟨*proof*⟩

lemma *fset-img-uneq-prod*: (λ(*t*,*e*). (*f t*, *e*)) |‘| *xs* ≠ *xs* ⟹ ∃ (*t*,*e*) ∈ *fset xs*. *f t* ≠ *t*
⟨*proof*⟩

lemma *contr-if-normalize1-uneq*:
normalize1 t1 ≠ *t1*
⟹ ∃ *v t2 e2*. *is-subtree* (*Node v* {|(*t2*,*e2*)|}) *t1* ∧ *rank* (*rev* (*root t2*)) < *rank* (*rev v*)
⟨*proof*⟩

lemma *contr-before-normalize1*:
⟦*is-subtree* (*Node v* {|(*t1*,*e1*)|}) (*normalize1 t3*); *rank* (*rev* (*root t1*)) < *rank* (*rev v*)⟧
⟹ ∃ *v'* *t2 e2*. *is-subtree* (*Node v'* {|(*t2*,*e2*)|}) *t3* ∧ *rank* (*rev* (*root t2*)) < *rank*

(*rev v'*)
⟨*proof*⟩

9.3.3 Normalizing Preserves Well-Formedness

lemma *normalize1-darcs-sub*: *darcs* (*normalize1 t1*) ⊆ *darcs t1*
⟨*proof*⟩

lemma *disjoint-darcs-normalize1*:
wf-darcs t1 ⟹ *disjoint-darcs* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |`| (*sucs t1*))
⟨*proof*⟩

lemma *wf-darcs-normalize1*: *wf-darcs t1* ⟹ *wf-darcs* (*normalize1 t1*)
⟨*proof*⟩

lemma *normalize1-dlverts-eq*[*simp*]: *dlverts* (*normalize1 t1*) = *dlverts t1*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *disjoint-dlverts-normalize1*:
disjoint-dlverts xs ⟹ *disjoint-dlverts* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |`| *xs*)
⟨*proof*⟩

lemma *disjoint-dlverts-normalize1-sucs*:
disjoint-dlverts (*sucs t1*) ⟹ *disjoint-dlverts* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |`| (*sucs t1*))
⟨*proof*⟩

lemma *disjoint-dlverts-normalize1-wf*:
wf-dlverts t1 ⟹ *disjoint-dlverts* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |`| (*sucs t1*))
⟨*proof*⟩

lemma *disjoint-dlverts-normalize1-wf'*:

wf-dlverts (*Node r xs*) $\Longrightarrow$ *disjoint-dlverts* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs*)
⟨*proof*⟩

lemma *root-empty-inter-dlverts-normalize1*:
assumes *wf-dlverts t1* **and** (*x1*,*e1*) $\in$ *fset* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| (*sucs t1*))
shows *set* (*root t1*) $\cap$ *dlverts x1* = {}
⟨*proof*⟩

lemma *wf-dlverts-normalize1*: *wf-dlverts t1* $\Longrightarrow$ *wf-dlverts* (*normalize1 t1*)
⟨*proof*⟩

corollary *list-dtree-normalize1*: *list-dtree* (*normalize1 t*)
⟨*proof*⟩

corollary *ranked-dtree-normalize1*: *ranked-dtree* (*normalize1 t*) *cmp*
⟨*proof*⟩

lemma *normalize-darcs-sub*: *darcs* (*normalize t1*) $\subseteq$ *darcs t1*
⟨*proof*⟩

lemma *normalize-dlverts-eq*: *dlverts* (*normalize t1*) = *dlverts t1*
⟨*proof*⟩

theorem *ranked-dtree-normalize*: *ranked-dtree* (*normalize t*) *cmp*
⟨*proof*⟩

9.3.4 Distinctness and hd preserved

lemma *distinct-normalize1*: ⟦$\forall$ *v*$\in$*dverts t*. *distinct v*; *v*$\in$*dverts* (*normalize1 t*)⟧ $\Longrightarrow$ *distinct v*
⟨*proof*⟩

lemma *distinct-normalize*: $\forall$ *v*$\in$*dverts t*. *distinct v* $\Longrightarrow$ $\forall$ *v*$\in$*dverts* (*normalize t*). *distinct v*
⟨*proof*⟩

lemma *normalize1-hd-root-eq*[*simp*]:
assumes *root t1* $\neq$ []
shows *hd* (*root* (*normalize1 t1*)) = *hd* (*root t1*)
⟨*proof*⟩

corollary *normalize1-hd-root-eq′*:
wf-dlverts t1 $\Longrightarrow$ *hd* (*root* (*normalize1 t1*)) = *hd* (*root t1*)
⟨*proof*⟩

lemma *normalize1-root-nempty*:
assumes *root t1* $\neq$ []
shows *root* (*normalize1 t1*) $\neq$ []

⟨*proof*⟩

lemma *normalize-hd-root-eq*[*simp*]: *root t1* ≠ [] ⟹ *hd* (*root* (*normalize t1*)) = *hd* (*root t1*)
⟨*proof*⟩

corollary *normalize-hd-root-eq'*[*simp*]: *wf-dlverts t1* ⟹ *hd* (*root* (*normalize t1*)) = *hd* (*root t1*)
⟨*proof*⟩

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*⟧
⟹ *t2* ≠ *normalize1 t2*
⟨*proof*⟩

lemma *sorted-ranks-if-normalize1-eq*:
⟦*wf-darcs t2*; *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2*; *t2* = *normalize1 t2*⟧
⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))
⟨*proof*⟩

lemma *normalize-sorted-ranks*:
⟦*is-subtree* (*Node r* {|(*t1*,*e1*)|}) (*normalize t*)⟧ ⟹ *rank* (*rev r*) ≤ *rank* (*rev* (*root t1*))
⟨*proof*⟩

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 Equiv)
⟨*proof*⟩

lemma *dtree-to-list-sorted-if-no-contr*:
⟦⋀*r1 t1 e1*. *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2* ⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))⟧
⟹ *sorted cmp''* (*dtree-to-list* (*Node r* {|(*t2*,*e2*)|}))
⟨*proof*⟩

lemma *dtree-to-list-sorted-if-no-contr'*:
⟦⋀*r1 t1 e1*. *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2* ⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))⟧
⟹ *sorted cmp''* (*dtree-to-list t2*)
⟨*proof*⟩

lemma *dtree-to-list-sorted-if-subtree*:
⟦*is-subtree t1 t2*;
⋀*r1 t1 e1*. *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2* ⟹ *rank* (*rev r1*) ≤ *rank* (*rev*

(*root t1*))⟧
⟹ *sorted cmp''* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|}))
⟨*proof*⟩

lemma *dtree-to-list-sorted-if-subtree'*:
⟦*is-subtree t1 t2*;
⋀*r1 t1 e1*. *is-subtree* (*Node r1* {|(*t1*,*e1*)|}) *t2* ⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*root t1*))⟧
⟹ *sorted cmp''* (*dtree-to-list t1*)
⟨*proof*⟩

lemma *normalize-dtree-to-list-sorted*:
is-subtree t1 (*normalize t*) ⟹ *sorted cmp''* (*dtree-to-list* (*Node r* {|(*t1*,*e1*)|}))
⟨*proof*⟩

lemma *normalize-dtree-to-list-sorted'*:
is-subtree t1 (*normalize t*) ⟹ *sorted cmp''* (*dtree-to-list t1*)
⟨*proof*⟩

lemma *gt-if-rank-contr*: *rank* (*rev r0*) < *rank* (*rev r*) ⟹ *compare cmp''* (*r*, *e*) (*r0*, *e0*) = *Greater*
⟨*proof*⟩

lemma *rank-le-if-ngt*: *compare cmp''* (*r*, *e*) (*r0*, *e0*) ≠ *Greater* ⟹ *rank* (*rev r*) ≤ *rank* (*rev r0*)
⟨*proof*⟩

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*⟩

lemma *cmp'-gt-if-cmp''-gt*: *compare cmp''* *x y* = *Greater* ⟹ *compare cmp'* *x y* = *Greater*
⟨*proof*⟩

lemma *cmp'-lt-if-cmp''-lt*: *compare cmp''* *x y* = *Less* ⟹ *compare cmp'* *x y* = *Less*
⟨*proof*⟩

lemma *cmp''-ge-if-cmp'-gt*:
compare cmp' *x y* = *Greater* ⟹ *compare cmp''* *x y* = *Greater* ∨ *compare cmp''* *x y* = *Equiv*
⟨*proof*⟩

lemma *cmp''-nlt-if-cmp'-gt*: *compare cmp'* *x y* = *Greater* ⟹ *compare cmp''* *y x* ≠ *Greater*
⟨*proof*⟩

interpretation *Comm*: *comp-fun-commute merge-f r xs* ⟨*proof*⟩

lemma *sorted-cmp''-merge*:
⟦*sorted cmp'' xs*; *sorted cmp'' ys*⟧ ⟹ *sorted cmp''* (*Sorting-Algorithms.merge cmp' xs ys*)
⟨*proof*⟩

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*⟩

lemma *not-single-subtree-if-nwf*:
¬*list-dtree* (*Node r xs*) ⟹ ¬*is-subtree* (*Node r1* {|(*t1*,*e1*)|}) (*merge* (*Node r xs*))
⟨*proof*⟩

lemma *not-single-subtree-if-nwf-sucs*:
¬*list-dtree t2* ⟹ ¬*is-subtree* (*Node r1* {|(*t1*,*e1*)|}) (*merge t2*)
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))

⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *merge1-dlverts-eq*[*simp*]: *dlverts* (*merge1 t*) = *dlverts t*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *merge1-dverts-sub*: *dverts* (*merge1 t1*) ⊆ *dverts t1*
⟨*proof*⟩

lemma *disjoint-dlverts-merge1*: *disjoint-dlverts* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| (*sucs t*))
⟨*proof*⟩

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*⟩

lemma *wf-dlverts-merge1*: *wf-dlverts* (*merge1 t*)
⟨*proof*⟩

lemma *merge1-darcs-eq*[*simp*]: *darcs* (*merge1 t*) = *darcs t*
⟨*proof*⟩

lemma *disjoint-darcs-merge1*: *disjoint-darcs* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| (*sucs t*))
⟨*proof*⟩

lemma *wf-darcs-merge1*: *wf-darcs* (*merge1 t*)
⟨*proof*⟩

theorem *ranked-dtree-merge1*: *ranked-dtree* (*merge1 t*) *cmp*
⟨*proof*⟩

lemma *distinct-merge1*:
⟦∀ *v*∈*dverts t*. *distinct v*; *v*∈*dverts* (*merge1 t*)⟧ ⟹ *distinct v*
⟨*proof*⟩

lemma *merge1-root-eq*[*simp*]: *root* (*merge1 t1*) = *root t1*
⟨*proof*⟩

lemma *merge1-hd-root-eq*[*simp*]: *hd* (*root* (*merge1 t1*)) = *hd* (*root t1*)
⟨*proof*⟩

lemma *merge1-mdeg-le*: *max-deg* (*merge1 t1*) ≤ *max-deg t1*
⟨*proof*⟩

lemma *merge1-childdeg-gt1-if-fcard-gt1*:
fcard (*sucs* (*merge1 t1*)) > 1 ⟹ ∃ *t* ∈ *fst* ‘ *fset* (*sucs t1*). *max-deg t* > 1
⟨*proof*⟩

lemma *merge1-fcard-le*: *fcard* (*sucs* (*merge1* (*Node r xs*))) ≤ *fcard xs*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *merge1-img-eq*: ∀ (*t2*,*e2*) ∈ *fset xs*. *merge1 t2* = *t2* ⟹ ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs*) = *xs*
⟨*proof*⟩

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*⟩

lemma *merge1-mdeg-gt1-if-uneq*:
assumes *merge1 t1* ≠ *t1*
shows *max-deg t1* > *1*
⟨*proof*⟩

corollary *merge1-eq-if-mdeg-le1*: *max-deg t1* ≤ *1* ⟹ *merge1 t1* = *t1*
⟨*proof*⟩

lemma *merge1-not-merge-if-fcard-gt1*:
⟦*merge1* (*Node r ys*) = *Node r xs*; *fcard xs* > *1*⟧ ⟹ *merge* (*Node r ys*) ≠ *Node r xs*
⟨*proof*⟩

lemma *merge1-img-if-not-merge*:
merge1 (*Node r xs*) ≠ *merge* (*Node r xs*)
⟹ *merge1* (*Node r xs*) = *Node r* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs*)
⟨*proof*⟩

lemma *merge1-img-if-fcard-gt1*:
⟦*merge1* (*Node r ys*) = *Node r xs*; *fcard xs* > *1*⟧
⟹ *merge1* (*Node r ys*) = *Node r* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *ys*)
⟨*proof*⟩

lemma *merge1-elem-in-img-if-fcard-gt1*:
⟦*merge1* (*Node r ys*) = *Node r xs*; *fcard xs* > *1*; (*t2*,*e2*) ∈ *fset xs*⟧
⟹ ∃ *t1*. (*t1*,*e2*) ∈ *fset ys* ∧ *merge1 t1* = *t2*
⟨*proof*⟩

lemma *child-mdeg-gt1-if-sub-fcard-gt1*:
⟦*is-subtree* (*Node r xs*) (*Node v ys*); *Node r xs* ≠ *Node v ys*; *fcard xs* > *1*⟧
⟹ ∃ *t1 e2*. (*t1*,*e2*) ∈ *fset ys* ∧ *max-deg t1* > *1*
⟨*proof*⟩

lemma *merge1-subtree-if-mdeg-gt1*:
⟦*is-subtree* (*Node r xs*) (*merge1 t1*); *max-deg* (*Node r xs*) > *1*⟧
⟹ ∃ *ys*. *merge1* (*Node r ys*) = *Node r xs* ∧ *is-subtree* (*Node r ys*) *t1*
⟨*proof*⟩

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*⟩

lemma *dverts-if-subtree-merge1*:
is-subtree (*Node r xs*) (*merge1 t1*) ⟹ *r* ∈ *dverts t1*
⟨*proof*⟩

lemma *subtree-merge1-orig*:
is-subtree (*Node r xs*) (*merge1 t1*) ⟹ ∃ *ys*. *is-subtree* (*Node r ys*) *t1*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *denormalize-set-eq-dlverts*: *max-deg t1* ≤ *1* ⟹ *set* (*denormalize t1*) = *dlverts t1*
⟨*proof*⟩

lemma *denormalize-set-sub-dlverts*: *set* (*denormalize t1*) ⊆ *dlverts t1*
⟨*proof*⟩

lemma *denormalize-distinct*:
⟦∀ *v* ∈ *dverts t1*. *distinct v*; *wf-dlverts t1*⟧ ⟹ *distinct* (*denormalize t1*)
⟨*proof*⟩

lemma *denormalize-hd-root*:
assumes *root t* ≠ []
shows *hd* (*denormalize t*) = *hd* (*root t*)
⟨*proof*⟩

lemma *denormalize-hd-root-wf*: *wf-dlverts t* $\Longrightarrow$ *hd* (*denormalize t*) = *hd* (*root t*)
⟨*proof*⟩

lemma *denormalize-nempty-if-wf*: *wf-dlverts t* $\Longrightarrow$ *denormalize t* $\neq$ []
⟨*proof*⟩

context *ranked-dtree*
begin

lemma *fcard-normalize-img-if-disjoint*:
disjoint-darcs xs $\Longrightarrow$ *fcard* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs*) = *fcard xs*
⟨*proof*⟩

lemma *fcard-merge1-img-if-disjoint*:
disjoint-darcs xs $\Longrightarrow$ *fcard* ((λ(*t*,*e*). (*merge1 t*,*e*)) |‘| *xs*) = *fcard xs*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *normalize1-dlverts-nempty*:
∀(*t*, *e*)∈*fset xs*. *dlverts t* $\neq$ {}
$\Longrightarrow$ ∀(*t*, *e*)∈*fset* ((λ(*t*, *e*). (*normalize1 t*, *e*)) |‘| *xs*). *dlverts t* $\neq$ {}
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fcard-normalize-img-if-wf-dlverts*:
wf-dlverts (*Node r xs*) $\Longrightarrow$ *fcard* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs*) = *fcard xs*
⟨*proof*⟩

lemma *fcard-normalize-img-if-wf-dlverts-sucs*:
wf-dlverts t1 $\Longrightarrow$ *fcard* ((λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| (*sucs t1*)) = *fcard* (*sucs t1*)
⟨*proof*⟩

lemma *singleton-normalize1*:
assumes *disjoint-darcs xs* **and** ∀*x*. *xs* $\neq$ {|*x*|}
shows ∀*x*. (λ(*t*,*e*). (*normalize1 t*,*e*)) |‘| *xs* $\neq$ {|*x*|}

⟨*proof*⟩

lemma *num-leaves-normalize1-eq*[*simp*]: *wf-darcs t1* ⟹ *num-leaves* (*normalize1 t1*) = *num-leaves t1*
⟨*proof*⟩

lemma *num-leaves-normalize-eq*[*simp*]: *wf-darcs t1* ⟹ *num-leaves* (*normalize t1*) = *num-leaves t1*
⟨*proof*⟩

lemma *num-leaves-normalize1-le*: *num-leaves* (*normalize1 t1*) ≤ *num-leaves t1*
⟨*proof*⟩

lemma *num-leaves-normalize-le*: *num-leaves* (*normalize t1*) ≤ *num-leaves t1*
⟨*proof*⟩

lemma *num-leaves-merge1-le*: *num-leaves* (*merge1 t1*) ≤ *num-leaves t1*
⟨*proof*⟩

lemma *num-leaves-merge1-lt*: *max-deg t1* > *1* ⟹ *num-leaves* (*merge1 t1*) < *num-leaves t1*
⟨*proof*⟩

lemma *ikkbz-num-leaves-decr*:
max-deg t1 > *1* ⟹ *num-leaves* (*merge1* (*normalize t1*)) < *num-leaves t1*
⟨*proof*⟩

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*)))
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *ikkbz-sub-darcs-sub*: *darcs* (*ikkbz-sub t*) ⊆ *darcs t*
⟨*proof*⟩

lemma *ikkbz-sub-dlverts-eq*[*simp*]: *dlverts* (*ikkbz-sub t*) = *dlverts t*
⟨*proof*⟩

lemma *ikkbz-sub-wf-darcs*: *wf-darcs* (*ikkbz-sub t*)
⟨*proof*⟩

lemma *ikkbz-sub-wf-dlverts*: *wf-dlverts* (*ikkbz-sub t*)
⟨*proof*⟩

theorem *ikkbz-sub-list-dtree*: *list-dtree* (*ikkbz-sub t*)
⟨*proof*⟩

corollary *ikkbz-sub-ranked-dtree*: *ranked-dtree* (*ikkbz-sub t*) *cmp*
⟨*proof*⟩

lemma *ikkbz-sub-mdeg-le1*: *max-deg* (*ikkbz-sub t1*) ≤ *1*
⟨*proof*⟩

corollary *denormalize-ikkbz-eq-dlverts*: *set* (*denormalize* (*ikkbz-sub t*)) = *dlverts t*
⟨*proof*⟩

lemma *distinct-ikkbz-sub*: ⟦∀ *v*∈*dverts t*. *distinct v*; *v*∈*dverts* (*ikkbz-sub t*)⟧ ⟹ *distinct v*
⟨*proof*⟩

corollary *distinct-denormalize-ikkbz-sub*:
∀ *v*∈*dverts t*. *distinct v* ⟹ *distinct* (*denormalize* (*ikkbz-sub t*))
⟨*proof*⟩

lemma *ikkbz-sub-hd-root*[*simp*]: *hd* (*root* (*ikkbz-sub t*)) = *hd* (*root t*)
⟨*proof*⟩

corollary *denormalize-ikkbz-sub-hd-root*[*simp*]: *hd* (*denormalize* (*ikkbz-sub t*)) = *hd* (*root t*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *to-list-dtree-wf-dverts*: *wf-dverts to-list-dtree*
⟨*proof*⟩

lemma *to-list-dtree-wf-dlverts*: *wf-dlverts to-list-dtree*
⟨*proof*⟩

lemma *to-list-dtree-wf-darcs*: *wf-darcs to-list-dtree*
⟨*proof*⟩

lemma *to-list-dtree-list-dtree*: *list-dtree to-list-dtree*
⟨*proof*⟩

lemma *to-list-dtree-ranked-dtree*: *ranked-dtree to-list-dtree cmp*
⟨*proof*⟩

interpretation *t*: *ranked-dtree to-list-dtree* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dverts-eq-verts-img*: *dverts to-list-dtree* = (λ*x*. [*x*]) ‘ *verts T*
⟨*proof*⟩

lemma *dlverts-eq-verts*: *dlverts to-list-dtree* = *verts T*
⟨*proof*⟩

theorem *ikkbz-set-eq-verts*: *set ikkbz-sub* = *verts T*
⟨*proof*⟩

lemma *distinct-to-list-tree*: ∀ *v*∈*verts to-list-tree*. *distinct v*
⟨*proof*⟩

lemma *distinct-to-list-dtree*: ∀ *v*∈*dverts to-list-dtree*. *distinct v*
⟨*proof*⟩

theorem *distinct-ikkbz-sub*: *distinct ikkbz-sub*
⟨*proof*⟩

lemma *to-list-dtree-root-eq-root*: *Dtree.root* (*to-list-dtree*) = [*root*]
⟨*proof*⟩

lemma *to-list-dtree-hd-root-eq-root*[*simp*]: *hd* (*Dtree.root to-list-dtree*) = *root*
⟨*proof*⟩

theorem *ikkbz-sub-hd-eq-root*[*simp*]: *hd ikkbz-sub* = *root*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *nempty-if-set-eq-verts*: *set xs* = *verts G* ⟹ *xs* ≠ []
⟨*proof*⟩

lemma *revorder-if-set-eq-verts*: *set xs* = *verts G* ⟹ *revorder* (*create-ldeep xs*) = *rev xs*
⟨*proof*⟩

lemma *cost-correct'*:
⟦*set xs* = *verts G*; *distinct xs*; *no-cross-products* (*create-ldeep xs*)⟧
⟹ *cost-r* (*hd xs*) (*rev xs*) = *cost-l xs*
⟨*proof*⟩

lemma *ikkbz-sub-verts-eq*: *r* ∈ *verts G* ⟹ *set* (*ikkbz-sub r*) = *verts G*
⟨*proof*⟩

lemma *ikkbz-sub-distinct*: *r* ∈ *verts G* ⟹ *distinct* (*ikkbz-sub r*)
⟨*proof*⟩

lemma *ikkbz-sub-hd-eq-root*: *r* ∈ *verts G* ⟹ *hd* (*ikkbz-sub r*) = *r*
⟨*proof*⟩

definition *ikkbz* :: 'a *list* **where**
ikkbz ≡ *arg-min-on cost-l* {*ikkbz-sub r*|*r*. *r* ∈ *verts G*}

lemma *ikkbz-sub-set-fin*: *finite* {*ikkbz-sub r*|*r*. *r* ∈ *verts G*}

⟨*proof*⟩

lemma *ikkbz-sub-set-nempty*: $\{ikkbz\text{-}sub\ r | r.\ r \in verts\ G\} \neq \{\}$
⟨*proof*⟩

lemma *ikkbz-in-ikkbz-sub-set*: $ikkbz \in \{ikkbz\text{-}sub\ r | r.\ r \in verts\ G\}$
⟨*proof*⟩

lemma *ikkbz-eq-ikkbz-sub*: $\exists\, r \in verts\ G.\ ikkbz = ikkbz\text{-}sub\ r$
⟨*proof*⟩

lemma *ikkbz-min-ikkbz-sub*: $r \in verts\ G \Longrightarrow cost\text{-}l\ ikkbz \leq cost\text{-}l\ (ikkbz\text{-}sub\ r)$
⟨*proof*⟩

lemma *ikkbz-distinct*: $distinct\ ikkbz$
⟨*proof*⟩

lemma *ikkbz-set-eq-verts*: $set\ ikkbz = verts\ G$
⟨*proof*⟩

lemma *ikkbz-nempty*: $ikkbz \neq []$
⟨*proof*⟩

lemma *ikkbz-hd-in-verts*: $hd\ ikkbz \in verts\ G$
⟨*proof*⟩

lemma *inorder-ikkbz*: $inorder\ (create\text{-}ldeep\ ikkbz) = ikkbz$
⟨*proof*⟩

lemma *inorder-ikkbz-distinct*: $distinct\ (inorder\ (create\text{-}ldeep\ ikkbz))$
⟨*proof*⟩

lemma *inorder-relations-eq-verts*: $relations\ (create\text{-}ldeep\ ikkbz) = verts\ G$
⟨*proof*⟩

theorem *ikkbz-valid-tree*: $valid\text{-}tree\ (create\text{-}ldeep\ ikkbz)$
⟨*proof*⟩

end

locale $old = list\text{-}dtree\ t$ **for** $t :: ('a\ list, 'b)\ dtree +$
fixes $rank :: 'a\ list \Rightarrow real$
begin

function $find\text{-}pos\text{-}aux :: 'a\ list \Rightarrow 'a\ list \Rightarrow ('a\ list, 'b)\ dtree \Rightarrow ('a\ list \times 'a\ list)$
where
$find\text{-}pos\text{-}aux\ v\ p\ (Node\ r\ \{|(t1, \text{-})|\}) =$

(*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*))
⟨*proof*⟩
termination ⟨*proof*⟩

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*)
⟨*proof*⟩
termination ⟨*proof*⟩

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*⟩

lemma *find-pos-rank-less*: *rank* (*rev v*) ≤ *rank* (*rev r*) ⟹ *find-pos-aux v p* (*Node r xs*) = (*p*,*r*)
⟨*proof*⟩

lemma *find-pos-y-in-dverts*: (*x*,*y*) = *find-pos-aux v p t1* ⟹ *y* ∈ *dverts t1*
⟨*proof*⟩

lemma *find-pos-x-in-dverts*: (*x*,*y*) = *find-pos-aux v p t1* ⟹ *x* ∈ *dverts t1* ∨ *p*=*x*
⟨*proof*⟩

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*) = ((∃ *y* ∈ *set xs*. *y* →$_T$ *x*) ∧ *forward-arcs xs*)

fun *no-back-arcs* :: *'a list* ⇒ *bool* **where**
  no-back-arcs [] = *True*
| *no-back-arcs* (*x*#*xs*) = ((∄ *y*. *y* ∈ *set xs* ∧ *y* →$_T$ *x*) ∧ *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*)
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *no-back-alt-aux*: (∀ *i j*. *i* ≥ *j* ∨ *j* ≥ *length xs* ∨ ¬(*xs*!*j* →$_T$ *xs*!*i*)) ⟹ *no-back xs*
  ⟨*proof*⟩

lemma *no-back-alt*: (∀ *i j*. *i* ≥ *j* ∨ *j* ≥ *length xs* ∨ ¬(*xs*!*j* →$_T$ *xs*!*i*)) ⟷ *no-back xs*
  ⟨*proof*⟩

lemma *no-back-arcs-alt-aux1*: ⟦*no-back-arcs xs*; *i* < *j*; *j* < *length xs*⟧ ⟹ ¬(*xs*!*j* →$_T$ *xs*!*i*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *no-back-insert*: *no-back* (*x*#*xs*) ⟹ *no-back xs*
⟨*proof*⟩

lemma *no-arc-fst-if-no-back*:
assumes *no-back* (*x*#*xs*) **and** *y* ∈ *set xs*
shows ¬ *y* →$_T$ *x*
⟨*proof*⟩

lemma *no-back-arcs-alt-aux2*: *no-back xs* ⟹ *no-back-arcs xs*
⟨*proof*⟩

lemma *no-back-arcs-alt*: *no-back xs* ⟷ *no-back-arcs xs*
⟨*proof*⟩

lemma *forward-arcs-alt-aux1*:
⟦*forward-arcs xs*; *i* ∈ {*1*..(*length* (*rev xs*) − *1*)}⟧ ⟹ ∃ *j* < *i*. (*rev xs*)!*j* →$_T$ (*rev xs*)!*i*
⟨*proof*⟩

lemma *forward-split-aux*:
assumes *forward* (*xs*@*ys*) **and** *i*∈{*1*..*length xs* − *1*}
shows ∃*j*<*i*. *xs*!*j* →$_T$ *xs*!*i*
⟨*proof*⟩

lemma *forward-split*: *forward* (*xs*@*ys*) ⟹ *forward xs*
⟨*proof*⟩

lemma *forward-cons*:
forward (*rev* (*x*#*xs*)) ⟹ *forward* (*rev xs*)
⟨*proof*⟩

lemma *arc-to-lst-if-forward*:
assumes *forward* (*rev* (*x*#*xs*)) **and** *xs* = *y*#*ys*
shows ∃ *y* ∈ *set xs*. *y* →$_T$ *x*
⟨*proof*⟩

lemma *forward-arcs-alt-aux2*: *forward* (*rev xs*) ⟹ *forward-arcs xs*
⟨*proof*⟩

lemma *forward-arcs-alt*: *forward xs* ⟷ *forward-arcs* (*rev xs*)
⟨*proof*⟩

corollary *forward-arcs-alt′*: *forward* (*rev xs*) ⟷ *forward-arcs xs*

⟨*proof*⟩

corollary *forward-arcs-split*: *forward-arcs* (*ys*@*xs*) ⟹ *forward-arcs xs*
⟨*proof*⟩

lemma *seq-conform-alt*: *seq-conform xs* ⟷ *forward xs* ∧ *no-back xs*
⟨*proof*⟩

lemma *forward-app-aux*:
assumes *forward s1 forward s2* ∃ *x*∈*set s1*. *x* →$_T$ *hd s2 i*∈{*1*..*length* (*s1*@*s2*) − *1*}
shows ∃ *j*<*i*. (*s1*@*s2*)!*j* →$_T$ (*s1*@*s2*)!*i*
⟨*proof*⟩

lemma *forward-app*: ⟦*forward s1*; *forward s2*; ∃ *x*∈*set s1*. *x* →$_T$ *hd s2*⟧ ⟹ *forward* (*s1*@*s2*)
⟨*proof*⟩

lemma *before-conform1I*: *before s1 s2* ⟹ *seq-conform s1*
⟨*proof*⟩

lemma *before-forward1I*: *before s1 s2* ⟹ *forward s1*
⟨*proof*⟩

lemma *before-no-back1I*: *before s1 s2* ⟹ *no-back s1*
⟨*proof*⟩

lemma *before-ArcI*: *before s1 s2* ⟹ ∃ *x* ∈ *set s1*. ∃ *y* ∈ *set s2*. *x* →$_T$ *y*
⟨*proof*⟩

lemma *before-conform2I*: *before s1 s2* ⟹ *seq-conform s2*
⟨*proof*⟩

lemma *before-forward2I*: *before s1 s2* ⟹ *forward s2*
⟨*proof*⟩

lemma *before-no-back2I*: *before s1 s2* ⟹ *no-back s2*
⟨*proof*⟩

lemma *hd-reach-all-forward-arcs*:
⟦*hd* (*rev xs*) ∈ *verts T*; *forward-arcs xs*; *x* ∈ *set xs*⟧ ⟹ *hd* (*rev xs*) →$^*{}_T$ *x*
⟨*proof*⟩

lemma *hd-reach-all-forward*:
⟦*hd xs* ∈ *verts T*; *forward xs*; *x* ∈ *set xs*⟧ ⟹ *hd xs* →$^*{}_T$ *x*
⟨*proof*⟩

lemma *hd-in-verts-if-forward*: *forward* (*x*#*y*#*xs*) ⟹ *hd* (*x*#*y*#*xs*) ∈ *verts T*
⟨*proof*⟩

lemma *two-elems-if-length-gt1*: *length xs* $>$ *1* $\Longrightarrow$ $\exists$ *x y ys*. *x*#*y*#*ys*=*xs*
⟨*proof*⟩

lemma *hd-in-verts-if-forward′*: ⟦*length xs* $>$ *1*; *forward xs*⟧ $\Longrightarrow$ *hd xs* $\in$ *verts T*
⟨*proof*⟩

lemma *hd-reach-all-forward′*:
⟦*length xs* $>$ *1*; *forward xs*; *x* $\in$ *set xs*⟧ $\Longrightarrow$ *hd xs* $\rightarrow^*{}_T$ *x*
⟨*proof*⟩

lemma *hd-reach-all-forward″*:
⟦*forward* (*x*#*y*#*xs*); *z* $\in$ *set* (*x*#*y*#*xs*)⟧ $\Longrightarrow$ *hd* (*x*#*y*#*xs*) $\rightarrow^*{}_T$ *z*
⟨*proof*⟩

lemma *no-back-if-distinct-forward*: ⟦*forward xs*; *distinct xs*⟧ $\Longrightarrow$ *no-back xs*
⟨*proof*⟩

corollary *seq-conform-if-dstnct-fwd*: ⟦*forward xs*; *distinct xs*⟧ $\Longrightarrow$ *seq-conform xs*
⟨*proof*⟩

lemma *forward-arcs-single*: *forward-arcs* [*x*]
⟨*proof*⟩

lemma *forward-single*: *forward* [*x*]
⟨*proof*⟩

lemma *no-back-arcs-single*: *no-back-arcs* [*x*]
⟨*proof*⟩

lemma *no-back-single*: *no-back* [*x*]
⟨*proof*⟩

lemma *seq-conform-single*: *seq-conform* [*x*]
⟨*proof*⟩

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*⟩

corollary *forward-arc-to-head*:
⟦*forward ys*; *set xs* $\cap$ *set ys* $=$ {}; *x* $\in$ *set xs*; *y* $\in$ *set ys*; *x* $\rightarrow_T$ *y*⟧
$\Longrightarrow$ *y* $=$ *hd ys*
⟨*proof*⟩

lemma *forward-app′*:
⟦*forward s1*; *forward s2*; *set s1* $\cap$ *set s2* $=$ {}; $\exists$ *x*$\in$*set s1*. $\exists$ *y*$\in$*set s2*. *x* $\rightarrow_T$ *y*⟧
$\Longrightarrow$ *forward* (*s1*@*s2*)

⟨*proof*⟩

lemma *reachable1-from-outside-dom*:
⟦$x \to^+_T y$; $x \notin$ *set ys*; $y \in$ *set ys*⟧ ⟹ ∃ x'. ∃ $y' \in$ *set ys*. $x' \notin$ *set ys* ∧ $x' \to_T y'$
⟨*proof*⟩

lemma *hd-reachable1-from-outside'*:
⟦$x \to^+_T y$; *forward ys*; $x \notin$ *set ys*; $y \in$ *set ys*⟧ ⟹ ∃ $y' \in$ *set ys*. $x \to^+_T$ *hd ys*
⟨*proof*⟩

lemma *hd-reachable1-from-outside*:
⟦$x \to^+_T y$; *forward ys*; *set xs* ∩ *set ys* = {}; $x \in$ *set xs*; $y \in$ *set ys*⟧
⟹ ∃ $y' \in$ *set ys*. $x \to^+_T$ *hd ys*
⟨*proof*⟩

lemma *reachable1-append-old-if-arc*:
assumes ∃ $x \in$ *set xs*. ∃ $y \in$ *set ys*. $x \to_T y$
and $z \notin$ *set xs*
and *forward xs*
and $y \in$ *set* (*xs* @ *ys*)
and $z \to^+_T y$
shows ∃ $y \in$ *set ys*. $z \to^+_T y$
⟨*proof*⟩

lemma *reachable1-append-old-if-arcU*:
⟦∃ $x \in$ *set xs*. ∃ $y \in$ *set ys*. $x \to_T y$; *set U* ∩ *set xs* = {}; $z \in$ *set U*;
forward xs; $y \in$ *set* (*xs* @ *ys*); $z \to^+_T y$⟧
⟹ ∃ $y \in$ *set ys*. $z \to^+_T y$
⟨*proof*⟩

lemma *before-arc-to-hd*: *before xs ys* ⟹ ∃ $x \in$ *set xs*. $x \to_T$ *hd ys*
⟨*proof*⟩

lemma *no-back-backarc-app1*:
⟦$j <$ *length* (*xs*@*ys*); $j \geq$ *length xs*; $i < j$; *no-back ys*; (*xs*@*ys*)!$j \to_T$ (*xs*@*ys*)!i⟧
⟹ $i <$ *length xs*
⟨*proof*⟩

lemma *no-back-backarc-app2*: ⟦*no-back xs*; $i < j$; (*xs*@*ys*)!$j \to_T$ (*xs*@*ys*)!i⟧ ⟹ $j \geq$ *length xs*
⟨*proof*⟩

lemma *no-back-backarc-i-in-xs*:
⟦*no-back ys*; $j <$ *length* (*xs*@*ys*); $i < j$; (*xs*@*ys*)!$j \to_T$ (*xs*@*ys*)!i⟧
⟹ *xs*!$i \in$ *set xs* ∧ (*xs*@*ys*)!i = *xs*!i
⟨*proof*⟩

lemma *no-back-backarc-j-in-ys*:
⟦*no-back xs*; $j <$ *length* (*xs*@*ys*); $i < j$; (*xs*@*ys*)!$j \to_T$ (*xs*@*ys*)!i⟧

$\Longrightarrow$ $ys!(j-length\ xs) \in set\ ys \wedge (xs@ys)!j = ys!(j-length\ xs)$
⟨*proof*⟩

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 $\exists x \in set\ xs.\ \exists y \in set\ ys.\ y \rightarrow_T x$
⟨*proof*⟩

lemma *no-back-backarc-difsets′*:
$[\![no\text{-}back\ xs;\ no\text{-}back\ ys;\ \exists i\ j.\ i < j \wedge j < length\ (xs@ys) \wedge (xs@ys)!j \rightarrow_T (xs@ys)!i]\!]$
$\Longrightarrow \exists x \in set\ xs.\ \exists y \in set\ ys.\ y \rightarrow_T x$
⟨*proof*⟩

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* $(xs\ @\ ys)$
⟨*proof*⟩

lemma *no-back-before*: $before\ xs\ ys \Longrightarrow no\text{-}back\ (xs@ys)$
⟨*proof*⟩

lemma *seq-conform-if-before*: $before\ xs\ ys \Longrightarrow seq\text{-}conform\ (xs@ys)$
⟨*proof*⟩

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*⟩

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*⟩

lemma *split-length-i*: $i \leq length\ bs \Longrightarrow \exists xs\ ys.\ xs@ys = bs \wedge length\ xs = i$
⟨*proof*⟩

lemma *split-length-i-prefix*:
assumes $length\ as \leq i\ i < length\ (as@bs)$
shows $\exists xs\ ys.\ xs@ys = bs \wedge length\ (as@xs) = i$
⟨*proof*⟩

lemma *forward-alt-aux1*:
assumes $i \in \{1..length\ xs - 1\}$ **and** $j{<}i$ **and** $xs!j \rightarrow_T xs!i$
shows $\exists as\ bs.\ as@bs = xs \wedge length\ as = i \wedge (\exists x \in set\ as.\ x \rightarrow_T xs!i)$
⟨*proof*⟩

lemma *forward-alt-aux1′*:
forward xs
$\Longrightarrow \forall\, i \in \{1..length\ xs - 1\}.\ \exists\, as\ bs.\ as@bs = xs \wedge length\ as = i \wedge (\exists\, x \in set\ as.\ x \rightarrow_T xs!i)$
⟨*proof*⟩

lemma *forward-alt-aux2*:
$[\![as@bs = xs;\ length\ as = i;\ \exists\, x \in set\ as.\ x \rightarrow_T xs!i]\!] \Longrightarrow \exists\, j{<}i.\ xs!j \rightarrow_T xs!i$
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *move-mid-forward-if-noarc*:
$[\![as \neq []; \neg(\exists\, x \in set\ U.\ \exists\, y \in set\ bs.\ x \rightarrow_T y);\ forward\ (as@U@bs@cs)]\!]$
$\Longrightarrow$ *forward* $(as@bs@U@cs)$
⟨*proof*⟩

lemma *move-mid-backward-if-noarc-aux*:
assumes $\exists\, x{\in}set\ U.\ x \rightarrow_T hd\ V$
and *forward V*
and *forward* $(as@U@bs@V@cs)$
and $i \in \{1..length\ (as@U@V@bs@cs) - 1\}$
shows $\exists j{<}i.\ (as@U@V@bs@cs)\ !\ j \rightarrow_T (as@U@V@bs@cs)\ !\ i$
⟨*proof*⟩

lemma *move-mid-backward-if-noarc*:
$[\![before\ U\ V;\ forward\ (as@U@bs@V@cs)]\!] \Longrightarrow forward\ (as@U@V@bs@cs)$
⟨*proof*⟩

lemma *move-mid-backward-if-noarc′*:

⟦∃ x∈set U. ∃ y∈set V. x →$_T$ y; forward V; set U ∩ set V = {}; forward (as@U@bs@V@cs)⟧
⟹ forward (as@U@V@bs@cs)
⟨proof⟩

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
⟨proof⟩

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⟩

lemma *list-empty-if-subset-dsjnt*: ⟦set xs ⊆ set ys; set xs ∩ set ys = {}⟧ ⟹ xs = []
⟨proof⟩

lemma *empty-if-sublist-dsjnt*: ⟦sublist xs ys; set xs ∩ set ys = {}⟧ ⟹ xs = []
⟨proof⟩

lemma *sublist-snd-if-fst-dsjnt*:
assumes sublist U (V@B) **and** set U ∩ set V = {}
shows sublist U B
⟨proof⟩

lemma *sublist-fst-if-snd-dsjnt*:
assumes sublist U (B@V) **and** set U ∩ set V = {}
shows sublist U B
⟨proof⟩

lemma *sublist-app*: sublist (A @ B) C ⟹ sublist A C ∧ sublist B C
⟨proof⟩

lemma *sublist-Cons*: sublist (A # B) C ⟹ sublist [A] C ∧ sublist B C
⟨proof⟩

lemma *sublist-set-elem*: ⟦sublist xs (A@B); x ∈ set xs⟧ ⟹ x ∈ set A ∨ x ∈ set B
⟨proof⟩

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⟩

lemma *suffix-ndjsnt-snd-if-nempty*: $[\![suffix\ xs\ (A@V);\ V \neq [];\ xs \neq []]\!] \Longrightarrow set\ xs \cap set\ V \neq \{\}$
⟨*proof*⟩

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*⟩

lemma *sublist-Y-cases-UV*:
assumes $\forall xs \in Y.\ \forall ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
and $U \in Y$
and $V \in Y$
and $U \neq []$
and $V \neq []$
and $(\forall xs \in Y.\ sublist\ xs\ (as@U@bs@V@cs))$
and $xs \in Y$
shows *sublist xs as ∨ sublist xs bs ∨ sublist xs cs ∨ U = xs ∨ V = xs*
⟨*proof*⟩

lemma *sublist-behind-if-nbefore*:
assumes *sublist U xs sublist V xs* $\nexists as\ bs\ cs.\ as @ U @ bs @ V @ cs = xs$ *set U ∩ set V = {}*
shows $\exists as\ bs\ cs.\ as @ V @ bs @ U @ cs = xs$
⟨*proof*⟩

lemma *sublists-preserv-move-U*:
$[\![set\ xs \cap set\ U = \{\};\ set\ xs \cap set\ V = \{\};\ V \neq [];\ sublist\ xs\ (as@U@bs@V@cs)]\!]$
$\Longrightarrow sublist\ xs\ (as@bs@U@V@cs)$
⟨*proof*⟩

lemma *sublists-preserv-move-UY*:
$[\![\forall xs \in Y.\ \forall ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\};\ xs \in Y;\ U \in Y;\ V \in Y;$
$V \neq [];\ sublist\ xs\ (as@U@bs@V@cs)]\!]$
$\Longrightarrow sublist\ xs\ (as@bs@U@V@cs)$
⟨*proof*⟩

lemma *sublists-preserv-move-UY-all*:
$[\![\forall xs \in Y.\ \forall ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\};\ U \in Y;\ V \in Y;$
$V \neq [];\ \forall xs \in Y.\ sublist\ xs\ (as@U@bs@V@cs)]\!]$
$\Longrightarrow \forall xs \in Y.\ sublist\ xs\ (as@bs@U@V@cs)$
⟨*proof*⟩

lemma *sublists-preserv-move-V*:
$[\![set\ xs \cap set\ U = \{\};\ set\ xs \cap set\ V = \{\};\ U \neq [];\ sublist\ xs\ (as@U@bs@V@cs)]\!]$
$\Longrightarrow sublist\ xs\ (as@U@V@bs@cs)$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *distinct-sublist-first*:
⟦*sublist as* (*x*#*xs*); *distinct* (*x*#*xs*); *x* ∈ *set as*⟧ ⟹ *take* (*length as*) (*x*#*xs*) = *as*
⟨*proof*⟩

lemma *distinct-sublist-first-remainder*:
⟦*sublist as* (*x*#*xs*); *distinct* (*x*#*xs*); *x* ∈ *set as*⟧ ⟹ *as* @ *drop* (*length as*) (*x*#*xs*)
= *x*#*xs*
⟨*proof*⟩

lemma *distinct-set-diff*: *distinct* (*xs*@*ys*) ⟹ *set ys* = *set* (*xs*@*ys*) − *set xs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *extract-length-decr*[*termination-simp*]:
List.extract P xs = *Some* (*as*,*x*,*bs*) ⟹ *length bs* < *length xs*
⟨*proof*⟩

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*⟩

lemma *separate-input-impl-none*: *separate-P P acc xs* = (*acc*,*xs*) ⟹ *List.extract*
P xs = *None*
⟨*proof*⟩

lemma *separate-input-iff-none*: *List.extract P xs = None* $\longleftrightarrow$ *separate-P P acc xs = (acc,xs)*
⟨*proof*⟩

lemma *separate-P-fst-acc*:
separate-P P acc xs = (as,bs) $\Longrightarrow$ $\exists$ *as'. as = as'@acc* $\wedge$ $(\forall$ *x* $\in$ *set as'. P x)*
⟨*proof*⟩

lemma *separate-P-fst*: *separate-P P [] xs = (as,bs)* $\Longrightarrow$ $\forall$ *x* $\in$ *set as. P x*
⟨*proof*⟩

10.2 Optimal Solution for Lists of Fixed Sets

lemma *distinct-seteq-set-length-eq*:
x $\in$ {*ys. set ys = xs* $\wedge$ *distinct ys*} $\Longrightarrow$ *length x = Finite-Set.card xs*
⟨*proof*⟩

lemma *distinct-seteq-set-Cons*:
⟦*Finite-Set.card xs = Suc n*; *x* $\in$ {*ys. set ys = xs* $\wedge$ *distinct ys*}⟧
$\Longrightarrow$ $\exists$ *y ys. y # ys = x* $\wedge$ *length ys = n* $\wedge$ *distinct ys* $\wedge$ *finite* (*set ys*)
⟨*proof*⟩

lemma *distinct-seteq-set-Cons'*:
⟦*Finite-Set.card xs = Suc n*; *x* $\in$ {*ys. set ys = xs* $\wedge$ *distinct ys*}⟧
$\Longrightarrow$ $\exists$ *y ys zs. y # ys = x* $\wedge$ *Finite-Set.card zs = n* $\wedge$ *distinct ys* $\wedge$ *set ys = zs*
⟨*proof*⟩

lemma *distinct-seteq-set-Cons''*:
⟦*Finite-Set.card xs = Suc n*; *x* $\in$ {*ys. set ys = xs* $\wedge$ *distinct ys*}⟧
$\Longrightarrow$ $\exists$ *y ys zs. y # ys = x* $\wedge$ *y* $\in$ *xs*
$\wedge$ *set ys = zs* $\wedge$ *Finite-Set.card zs = n* $\wedge$ *distinct ys* $\wedge$ *finite zs*
⟨*proof*⟩

lemma *distinct-seteq-set-Cons-in-set*:
⟦*Finite-Set.card xs = Suc n*; *x* $\in$ {*ys. set ys = xs* $\wedge$ *distinct ys*}⟧
$\Longrightarrow$ $\exists$ *y ys zs. y#ys = x* $\wedge$ *y* $\in$ *xs* $\wedge$ *Finite-Set.card zs = n* $\wedge$ *ys*$\in$*{ys. set ys = zs* $\wedge$ *distinct ys}*
⟨*proof*⟩

lemma *distinct-seteq-set-Cons-in-set'*:
⟦*Finite-Set.card xs = Suc n*; *x* $\in$ {*ys. set ys = xs* $\wedge$ *distinct ys*}⟧
$\Longrightarrow$ $\exists$ *y ys. x = y#ys* $\wedge$ *y* $\in$ *xs* $\wedge$ *ys*$\in${*ys. set ys = (xs* $-$ {*y*}*)* $\wedge$ *distinct ys*}
⟨*proof*⟩

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*}}

⟨*proof*⟩

lemma *distinct-seteq-sub-set-union*:
Finite-Set.card xs = *Suc n*
⟹ {*ys*. *set ys* = *xs* ∧ *distinct ys*}
⊆ {*y* # *ys* |*y ys*. *y* ∈ *xs* ∧ *ys* ∈ {*as*. ∃ *a* ∈ *xs*. *set as* = (*xs* − {*a*}) ∧ *distinct as*}}
⟨*proof*⟩

lemma *finite-set-union*: ⟦*finite ys*; ∀ *y* ∈ *ys*. *finite y*⟧ ⟹ *finite* (⋃ *y* ∈ *ys*. *y*)
⟨*proof*⟩

lemma *Cons-set-eq-union-set*:
{*x* # *y* | *x y y*′. *x* ∈ *xs* ∧ *y* ∈ *y*′ ∧ *y*′ ∈ *ys*} = {*x* # *y* | *x y*. *x* ∈ *xs* ∧ *y* ∈ (⋃ *y* ∈ *ys*. *y*)}
⟨*proof*⟩

lemma *finite-set-Cons-union-finite*:
⟦*finite xs*; *finite ys*; ∀ *y* ∈ *ys*. *finite y*⟧
⟹ *finite* {*x* # *y* | *x y*. *x* ∈ *xs* ∧ *y* ∈ (⋃ *y* ∈ *ys*. *y*)}
⟨*proof*⟩

lemma *finite-set-Cons-finite*:
⟦*finite xs*; *finite ys*; ∀ *y* ∈ *ys*. *finite y*⟧
⟹ *finite* {*x* # *y* | *x y y*′. *x* ∈ *xs* ∧ *y* ∈ *y*′ ∧ *y*′ ∈ *ys*}
⟨*proof*⟩

lemma *finite-set-Cons-finite*′:
⟦*finite xs*; *finite ys*⟧ ⟹ *finite* {*x* # *y* |*x y*. *x* ∈ *xs* ∧ *y* ∈ *ys*}
⟨*proof*⟩

lemma *Cons-set-alt*: {*x* # *y* |*x y*. *x* ∈ *xs* ∧ *y* ∈ *ys*} = {*zs*. ∃ *x y*. *x* # *y* = *zs* ∧ *x* ∈ *xs* ∧ *y* ∈ *ys*}
⟨*proof*⟩

lemma *Cons-set-sub*:
assumes *Finite-Set.card xs* = *Suc n*
shows {*ys*. *set ys* = *xs* ∧ *distinct ys*}
⊆ {*x* # *y* |*x y*. *x* ∈ *xs* ∧ *y* ∈ (⋃ *y* ∈ *xs*. {*as*. *set as* = *xs* − {*y*} ∧ *distinct as*})}
⟨*proof*⟩

lemma *distinct-seteq-finite*: *finite xs* ⟹ *finite* {*ys*. *set ys* = *xs* ∧ *distinct ys*}
⟨*proof*⟩

lemma *distinct-setsub-split*:
{*ys*. *set ys* ⊆ *xs* ∧ *distinct ys*}
= {*ys*. *set ys* = *xs* ∧ *distinct ys*} ∪ (⋃ *y* ∈ *xs*. {*ys*. *set ys* ⊆ (*xs*−{*y*}) ∧ *distinct ys*})
⟨*proof*⟩

lemma *valid-UV-lists-finite*:
finite xs $\Longrightarrow$ *finite* $\{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x\}$
⟨*proof*⟩

lemma *valid-UV-lists-r-subset*:
$\{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r]\}$
$\subseteq \{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x\}$
⟨*proof*⟩

lemma *valid-UV-lists-r-finite*:
finite xs $\Longrightarrow$ *finite* $\{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r]\}$
⟨*proof*⟩

lemma *valid-UV-lists-arg-min-ex-aux*:
$[\![$*finite ys*; $ys \neq \{\}$; $ys = \{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x\}]\!]$
$\Longrightarrow \exists\, y \in ys.\ \forall\, z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
⟨*proof*⟩

lemma *valid-UV-lists-arg-min-ex*:
$[\![$*finite xs*; $ys \neq \{\}$; $ys = \{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x\}]\!]$
$\Longrightarrow \exists\, y \in ys.\ \forall\, z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
⟨*proof*⟩

lemma *valid-UV-lists-arg-min-r-ex-aux*:
$[\![$*finite ys*; $ys \neq \{\}$;
$ys = \{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r]\}]\!]$
$\Longrightarrow \exists\, y \in ys.\ \forall\, z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
⟨*proof*⟩

lemma *valid-UV-lists-arg-min-r-ex*:
$[\![$*finite xs*; $ys \neq \{\}$;
$ys = \{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r]\}]\!]$
$\Longrightarrow \exists\, y \in ys.\ \forall\, z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
⟨*proof*⟩

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*⟩

lemma *valid-UV-lists-nemtpy′*:
$[\![$*finite xs*; *set* $U \cap$ *set* $V = \{\}$; *set* $U \subseteq xs$; *set* $V \subseteq xs$; *distinct* U; *distinct* $V]\!]$

$\Longrightarrow \{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x\} \neq \{\}$
⟨*proof*⟩

lemma *valid-UV-lists-nemtpy-r*:
assumes *finite xs* **and** *set* $(U@V) \subseteq xs$ **and** *distinct* $(U@V)$
and *take 1 U* $= [r] \vee r \notin set\ U \cup set\ V$ **and** $r \in xs$
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*⟩

lemma *valid-UV-lists-nemtpy-r′*:
⟦*finite xs*; *set U* $\cap$ *set V* $= \{\}$; *set U* $\subseteq$ *xs*; *set V* $\subseteq$ *xs*; *distinct U*; *distinct V*;
take 1 U $= [r] \vee r \notin set\ U \cup set\ V$; $r \in xs$⟧
$\Longrightarrow \{x.\ \exists\, as\ bs\ cs.\ as@U@bs@V@cs = x \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r]\} \neq \{\}$
⟨*proof*⟩

lemma *valid-UV-lists-arg-min-ex′*:
⟦*finite xs*; *set U* $\cap$ *set V* $= \{\}$; *set U* $\subseteq$ *xs*; *set V* $\subseteq$ *xs*; *distinct U*; *distinct V*;
ys $= \{x.\ (\exists\, as\ bs\ cs.\ as@U@bs@V@cs = x) \wedge set\ x = xs \wedge distinct\ x\}$⟧
$\Longrightarrow \exists\, y \in ys.\ \forall\, z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
⟨*proof*⟩

lemma *valid-UV-lists-arg-min-r-ex′*:
⟦*finite xs*; *set U* $\cap$ *set V* $= \{\}$; *set U* $\subseteq$ *xs*; *set V* $\subseteq$ *xs*; *distinct U*; *distinct V*;
take 1 U $= [r] \vee r \notin set\ U \cup set\ V$; $r \in xs$;
ys $= \{x.\ (\exists\, as\ bs\ cs.\ as@U@bs@V@cs = x) \wedge set\ x = xs \wedge distinct\ x \wedge take\ 1\ x = [r]\}$⟧
$\Longrightarrow \exists\, y \in ys.\ \forall\, z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
⟨*proof*⟩

lemma *valid-UV-lists-alt*:
assumes $P = (\lambda x.\ (\exists\, as\ bs\ cs.\ as@U@bs@V@cs = x) \wedge set\ x = xs \wedge distinct\ x)$
shows $\{x.\ (\exists\, as\ bs\ cs.\ as@U@bs@V@cs = x) \wedge set\ x = xs \wedge distinct\ x\} = \{ys.\ P\ ys\}$
⟨*proof*⟩

lemma *valid-UV-lists-argmin-ex*:
fixes *cost* :: *'a list* ⇒ *real*
assumes $P = (\lambda x.\ (\exists\, as\ bs\ cs.\ as@U@bs@V@cs = x) \wedge set\ x = xs \wedge distinct\ x)$
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*
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*⟩

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*))
⟨*proof*⟩

lemma *valid-UV-lists-argmin-r-ex*:
fixes *cost* :: *'a list* ⇒ *real*
assumes *P* = (λ*x*. (∃ *as bs cs*. *as*@*U*@*bs*@*V*@*cs* = *x*) ∧ *set x* = *xs* ∧ *distinct x* ∧ *take 1 x* = [*r*])
and *finite xs*
and *set U* ∩ *set V* = {}
and *set U* ⊆ *xs*
and *set V* ⊆ *xs*
and *distinct U*
and *distinct V*
and *take 1 U* = [*r*] ∨ *r* ∉ *set U* ∪ *set V*
and *r* ∈ *xs*
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*⟩

lemma *valid-UV-lists-argmin-r-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*
and *take 1 U* = [*r*] ∨ *r* ∉ *set U* ∪ *set V*
and *r* ∈ *xs*
shows ∃ *as' bs' cs'*. *set* (*as'* @ *U* @ *bs'* @ *V* @ *cs'*) = *xs*
∧ *distinct* (*as'* @ *U* @ *bs'* @ *V* @ *cs'*) ∧ *take 1* (*as'* @ *U* @ *bs'* @ *V* @ *cs'*) = [*r*]
∧ (∀ *as bs cs*. *set* (*as* @ *U* @ *bs* @ *V* @ *cs*) = *xs*
∧ *distinct* (*as* @ *U* @ *bs* @ *V* @ *cs*) ∧ *take 1* (*as* @ *U* @ *bs* @ *V* @ *cs*) = [*r*]

$\longrightarrow$ *cost* (*as'* @ *U* @ *bs'* @ *V* @ *cs'*) $\leq$ *cost* (*as* @ *U* @ *bs* @ *V* @ *cs*))
⟨*proof*⟩

lemma *valid-UV-lists-argmin-r-ex-noP'*:
fixes *cost* :: *'a list* $\Rightarrow$ *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*)))
⟨*proof*⟩

lemma *take1-split-nempty*: *ys* $\neq$ [] $\Longrightarrow$ *take 1* (*xs*@*ys*@*zs*) = *take 1* (*xs*@*ys*)
⟨*proof*⟩

lemma *take1-elem*: ⟦*take 1* (*xs*@*ys*) = [*r*]; *r* $\in$ *set xs*⟧ $\Longrightarrow$ *take 1 xs* = [*r*]
⟨*proof*⟩

lemma *take1-nelem*: ⟦*take 1* (*xs*@*ys*) = [*r*]; *r* $\notin$ *set ys*⟧ $\Longrightarrow$ *take 1 xs* = [*r*]
⟨*proof*⟩

lemma *take1-split-nelem-nempty*: ⟦*take 1* (*xs*@*ys*@*zs*) = [*r*]; *ys* $\neq$ []; *r* $\notin$ *set ys*⟧
$\Longrightarrow$ *take 1 xs* = [*r*]
⟨*proof*⟩

lemma *take1-empty-if-nelem*: ⟦*take 1* (*as*@*bs*@*cs*) = [*r*]; *r* $\notin$ *set as*⟧ $\Longrightarrow$ *as* = []
⟨*proof*⟩

lemma *take1-empty-if-mid*: ⟦*take 1* (*as*@*bs*@*cs*) = [*r*]; *r* $\in$ *set bs*; *distinct* (*as*@*bs*@*cs*)⟧
$\Longrightarrow$ *as* = []
⟨*proof*⟩

lemma *take1-mid-if-elem*:
⟦*take 1* (*as*@*bs*@*cs*) = [*r*]; *r* $\in$ *set bs*; *distinct* (*as*@*bs*@*cs*)⟧ $\Longrightarrow$ *take 1 bs* = [*r*]
⟨*proof*⟩

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 ∩ set V = {}*
and *set U ⊆ xs*
and *set V ⊆ xs*
and *distinct U*
and *distinct V*
and *r ∉ set U ∪ set V*
and *r ∈ xs*
shows ∃ *as' cs'. distinct (as' @ U @ V @ cs') ∧ take 1 (as' @ U @ V @ cs') = [r]*
∧ *set (as' @ U @ V @ cs') = xs ∧ (∀ as bs cs. set (as @ U @ bs @ V @ cs) = xs*
∧ *distinct (as @ U @ bs @ V @ cs) ∧ take 1 (as @ U @ bs @ V @ cs) = [r]*
⟶ *cost (rev (as' @ U @ V @ cs')) ≤ cost (rev (as @ U @ bs @ V @ cs)))*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *make-list-concat-rev-eq*: *concat (make-list-P P acc xs) = concat (rev acc) @ concat xs*
⟨*proof*⟩

lemma *combine-lists-sublists*:
∃ *x ∈ {y} ∪ set xs. sublist as x ⟹ ∃ x ∈ set (combine-lists-P P y xs). sublist as x*
⟨*proof*⟩

lemma *make-list-sublists*:
∃ *x ∈ set acc ∪ set xs. sublist cs x ⟹ ∃ x ∈ set (make-list-P P acc xs). sublist cs x*
⟨*proof*⟩

lemma *combine-lists-nempty*: ⟦*[] ∉ set xs; y ≠ []⟧ ⟹ [] ∉ set (combine-lists-P P y xs)*
⟨*proof*⟩

lemma *make-list-nempty*:
⟦[] ∉ *set acc*; [] ∉ *set xs*⟧ ⟹ [] ∉ *set* (*make-list-P P acc xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *combine-lists-single*: *xs* = [*x*] ⟹ *combine-lists-P P y xs* = [*x*@*y*]
⟨*proof*⟩

lemma *combine-lists-lastP*:
P (*last xs*) ⟹ (∃ *x*. *combine-lists-P P y xs* = [*x*]) ∨ (*P* (*last* (*combine-lists-P P y xs*)))
⟨*proof*⟩

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*⟩

corollary *make-list-notP-empty-acc*:
(∀ *x*∈*set* (*make-list-P P* [] *xs*). ¬*P x*) ∨ (∃ *y ys*. *make-list-P P* [] *xs* = *y* # *ys* ∧ *P y*)
⟨*proof*⟩

definition *unique-set-r* :: *'a* ⇒ *'a list set* ⇒ *'a list* ⇒ *bool* **where**
unique-set-r r Y ys ⟷ *set ys* = ⋃(*set* ‘ *Y*) ∧ *distinct ys* ∧ *take 1 ys* = [*r*]

context *directed-tree*
begin

definition *fwd-sub* :: *'a* ⇒ *'a list set* ⇒ *'a list* ⇒ *bool* **where**
fwd-sub r Y ys ⟷ *unique-set-r r Y ys* ∧ *forward ys* ∧ (∀ *xs* ∈ *Y*. *sublist xs ys*)

lemma *distinct-mid-unique1*: ⟦*distinct* (*xs*@*U*@*ys*); *U*≠[]; *xs*@*U*@*ys* = *as*@*U*@*bs*⟧
⟹ *as* = *xs*
⟨*proof*⟩

lemma *distinct-mid-unique2*: ⟦*distinct* (*xs*@*U*@*ys*); *U*≠[]; *xs*@*U*@*ys* = *as*@*U*@*bs*⟧
⟹ *ys* = *bs*
⟨*proof*⟩

lemma *concat-all-sublist*: ∀ *x* ∈ *set xs*. *sublist x* (*concat xs*)
⟨*proof*⟩

lemma *concat-all-sublist-rev*: $\forall\, x \in set\ xs.\ sublist\ x\ (concat\ (rev\ xs))$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *nonempty-notin-distinct-prefix*:
assumes *distinct* $(as@bs@V@cs)$ **and** $concat\ as' = as$ **and** $V \neq []$
shows $V \notin set\ as'$
⟨*proof*⟩

lemma *concat-split-UV*:
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 $V \in Y$
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 $U \neq []$
and $V \neq []$
shows $\exists as'\ bs'\ cs'.\ concat\ as' = as \wedge concat\ bs' = bs \wedge concat\ cs' = cs$
$\wedge\ set\ (as'@U\#bs'@V\#cs') = Y \wedge distinct\ (as'@U\#bs'@V\#cs')$
⟨*proof*⟩

lemma *cost-decr-if-noarc-lessrank*:
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 $concat\ (b\#bs') = bs$
and $(\forall xs \in Y.\ sublist\ xs\ as \vee sublist\ xs\ U$
$\vee\ (\exists x \in set\ (b\#bs').\ sublist\ xs\ x) \vee sublist\ xs\ cs)$
and $\neg(\exists x \in set\ U.\ \exists y \in set\ b.\ x \rightarrow_T y)$
and $rank\ (rev\ b) < rank\ (rev\ U)$
shows *fwd-sub r Y* $(as@b@U@concat\ bs'@cs)$
$\wedge\ cost\ (rev\ (as@b@U@concat\ bs'@cs)) < cost\ (rev\ (as@U@bs@cs))$
⟨*proof*⟩

lemma *cost-decr-if-noarc-lessrank'*:
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* $(b\#bs') = bs$
and $(\forall\, xs \in Y.\ sublist\ xs\ as \vee sublist\ xs\ U$
$\vee\ (\exists\, x \in set\ (b\#bs').\ sublist\ xs\ x) \vee sublist\ xs\ cs)$
and $\neg(\exists\, x \in set\ U.\ \exists\, y \in set\ b.\ x \rightarrow_T y)$
and *rank* $(rev\ b) <$ *rank* $(rev\ V)$
and *rank* $(rev\ V) \leq$ *rank* $(rev\ U)$
shows *fwd-sub* $r\ Y\ (as@b@U@concat\ bs'@cs)$
$\wedge$ *cost* $(rev\ (as@b@U@concat\ bs'@cs)) <$ *cost* $(rev\ (as@U@bs@cs))$
⟨*proof*⟩

lemma *sublist-exists-append*:
$\exists\, a{\in}set\ ((x\ \#\ xs)\ @\ [b]).\ sublist\ ys\ a \Longrightarrow \exists\, a{\in}set(xs\ @\ [x@b]).\ sublist\ ys\ a$
⟨*proof*⟩

lemma *sublist-set-concat-cases*:
$\exists\, a{\in}set\ ((x\ \#\ xs)\ @\ [b]).\ sublist\ ys\ a \Longrightarrow sublist\ ys\ (concat\ (rev\ xs)) \vee sublist\ ys\ x \vee sublist\ ys\ b$
⟨*proof*⟩

lemma *sublist-set-concat-or-cases-aux1*:
$sublist\ ys\ as \vee sublist\ ys\ U \vee sublist\ ys\ cs$
$\Longrightarrow sublist\ ys\ (as\ @\ U\ @\ concat\ (rev\ xs)) \vee sublist\ ys\ cs$
⟨*proof*⟩

lemma *sublist-set-concat-or-cases-aux2*:
$\exists\, a{\in}set\ ((x\ \#\ xs)\ @\ [b]).\ sublist\ ys\ a$
$\Longrightarrow sublist\ ys\ (as\ @\ U\ @\ concat\ (rev\ xs)) \vee sublist\ ys\ x \vee sublist\ ys\ b$
⟨*proof*⟩

lemma *sublist-set-concat-or-cases*:
$sublist\ ys\ as \vee sublist\ ys\ U \vee (\exists\, a{\in}set\ ((x\#xs)\ @\ [b]).\ sublist\ ys\ a) \vee sublist\ ys\ cs \Longrightarrow$
$sublist\ ys\ (as@U@\ concat\ (rev\ xs)) \vee sublist\ ys\ x \vee (\exists\, a{\in}set\ [b].\ sublist\ ys\ a) \vee sublist\ ys\ cs$
⟨*proof*⟩

corollary *not-reachable1-append-if-not-old*:
$[\![\neg\ (\exists\, z{\in}set\ U.\ \exists\, y{\in}set\ b.\ z \rightarrow^+_T y);\ set\ U \cap set\ x = \{\};\ forward\ x;$
$\exists\, z{\in}set\ x.\ \exists\, y{\in}set\ b.\ z \rightarrow_T y]\!]$
$\Longrightarrow \neg\ (\exists\, z{\in}set\ U.\ \exists\, y{\in}set\ (x@b).\ z \rightarrow^+_T y)$
⟨*proof*⟩

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 (∀ *xs* ∈ *Y*. *sublist xs as* ∨ *sublist xs U*
∨ (∃ *x* ∈ *set* (*ys* @ [*b*]). *sublist xs x*) ∨ *sublist xs cs*)
and *rank* (*rev V*) ≤ *rank* (*rev U*)
and ¬(∃ *x* ∈ *set U*. ∃ *y* ∈ *set b*. *x* →⁺ *T y*)
and *rank* (*rev b*) < *rank* (*rev V*)
and *P* = (λ*x*. *rank* (*rev x*) < *rank* (*rev V*))
and ∀ *x*∈*set ys*. ¬*P x*
and ∀ *xs*. *fwd-sub r Y xs* ⟶ *cost* (*rev* (*as*@*U*@*bs*@*cs*)) ≤ *cost* (*rev xs*)
and ∀ *x* ∈ *set ys*. *x* ≠ []
and ∀ *x* ∈ *set ys*. *forward x*
and *forward b*
shows ∀ *x*∈*set* (*combine-lists-P P b ys*). ¬*P x* ∧ *forward x*
⟨*proof*⟩

lemma *sublist-app-l*: *sublist ys cs* ⟹ *sublist ys* (*xs* @ *cs*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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 ∀ *xs*. *fwd-sub r Y xs* ⟶ *cost* (*rev* (*as*@*U*@*bs*@*cs*)) ≤ *cost* (*rev xs*)
and ∀ *x* ∈ *set ys*. *x* ≠ []
and ∀ *x* ∈ *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 $\forall x{\in}set\ (make\text{-}list\text{-}P\ P\ acc\ ys).\ \neg P\ x$
⟨*proof*⟩

lemma *no-back-reach1-if-fwd-dstct-bs*:
$[\![forward\ (as@concat\ bs@V@cs);\ distinct\ (as@concat\ bs@V@cs);\ xs \in set\ bs]\!]$
$\Longrightarrow \neg(\exists x'{\in}set\ V.\ \exists y{\in}set\ xs.\ x' \rightarrow^+_T y)$
⟨*proof*⟩

lemma *mid-ranks-ge-if-reach1*:
assumes $[] \notin Y$
and $U \in Y$
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 $\bigwedge xs.\ [\![xs \in Y;\ \exists y{\in}set\ xs.\ \neg(\exists x'{\in}set\ V.\ x' \rightarrow^+_T y) \wedge (\exists x{\in}set\ U.\ x \rightarrow^+_T y);\ xs \neq U]\!] \Longrightarrow rank\ (rev\ V) \leq rank\ (rev\ xs)$
shows $\forall xs \in set\ bs'.\ (\exists x{\in}set\ U.\ \exists y{\in}set\ xs.\ x \rightarrow^+_T y) \longrightarrow rank\ (rev\ V) \leq rank\ (rev\ xs)$
⟨*proof*⟩

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 $\forall zs.\ fwd\text{-}sub\ r\ Y\ zs \longrightarrow cost\ (rev\ (as@U@bs@V@cs)) \leq cost\ (rev\ zs)$
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*⟩

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* $(rev\ (as@ds@V@concat\ bs@cs)) \leq cost\ (rev\ (as@ds@concat\ bs@V@cs))$
⟨*proof*⟩

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* $(rev\ V) \leq rank\ (rev\ bs)$
⟨*proof*⟩

lemma *bs-ge-if-optimal*:
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 $r \notin set\ U$
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 *forward* $(as@U@bs@V@cs)$
and $bs \neq []$
and *rank* $(rev\ V) \leq rank\ (rev\ U)$
and $\forall zs.\ fwd\text{-}sub\ r\ Y\ zs \longrightarrow cost\ (rev\ (as@U@bs@V@cs)) \leq cost\ (rev\ zs)$
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 *rank* $(rev\ V) \leq rank\ (rev\ bs)$
⟨*proof*⟩

lemma *bs-ranks-only-ge-r*:
assumes $[] \notin Y$

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*⟩

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*⟩

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*⟩

lemma *forward-UV-lists-subset*:
{*x*. *set x* = *X* ∧ *distinct x* ∧ *take 1 x* = [*r*] ∧ *forward x* ∧ (∀ *xs* ∈ *Y*. *sublist xs x*)}
⊆ {*x*. *set x* = *X* ∧ *distinct x*}
⟨*proof*⟩

lemma *forward-UV-lists-finite*:
finite xs
⟹ *finite* {*x*. *set x* = *xs* ∧ *distinct x* ∧ *take 1 x* = [*r*] ∧ *forward x* ∧ (∀ *xs* ∈ *Y*. *sublist xs x*)}

⟨*proof*⟩

lemma *forward-UV-lists-arg-min-ex-aux*:
⟦*finite ys*; *ys* ≠ {};
ys = {*x*. *set x* = *xs* ∧ *distinct x* ∧ *take 1 x* = [*r*] ∧ *forward x* ∧ (∀ *xs* ∈ *Y*. *sublist xs x*)}⟧
⟹ ∃ *y* ∈ *ys*. ∀ *z* ∈ *ys*. (*f* :: *'a list* ⇒ *real*) *y* ≤ *f z*
⟨*proof*⟩

lemma *forward-UV-lists-arg-min-ex*:
⟦*finite xs*; *ys* ≠ {};
ys = {*x*. *set x* = *xs* ∧ *distinct x* ∧ *take 1 x* = [*r*] ∧ *forward x* ∧ (∀ *xs* ∈ *Y*. *sublist xs x*)}⟧
⟹ ∃ *y* ∈ *ys*. ∀ *z* ∈ *ys*. (*f* :: *'a list* ⇒ *real*) *y* ≤ *f z*
⟨*proof*⟩

lemma *forward-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* ∧ *forward ys* ∧ (∀ *xs* ∈ *Y*. *sublist xs ys*))
and ∃ *x*. *Q x*
shows ∃ *zs*. *Q zs* ∧ (∀ *as*. *Q as* ⟶ *f zs* ≤ *f as*)
⟨*proof*⟩

lemma *forward-UV-lists-argmin-ex*:
fixes *f* :: *'a list* ⇒ *real*
assumes ∃ *x*. *fwd-sub r Y x*
shows ∃ *zs*. *fwd-sub r Y zs* ∧ (∀ *as*. *fwd-sub r Y as* ⟶ *f zs* ≤ *f as*)
⟨*proof*⟩

lemma *no-gap-if-contr-seq-fwd*:
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 *rank* (*rev V*) ≤ *rank* (*rev U*)
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*)
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*⟩

lemma *combine-union-sets-alt*:

fixes X Y
defines $Z \equiv X \cup \{x.\ x \in Y \wedge set\ x \cap \bigcup(set\ `\ X) = \{\}\}$
assumes $\forall\, xs \in Y.\ \forall\, ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
and $\forall\, xs \in X.\ \forall\, ys \in X.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
shows $Z = X \cup (Y - \{x.\ set\ x \cap \bigcup(set\ `\ X) \neq \{\}\})$
⟨*proof*⟩

lemma *combine-union-sets-disjoint*:
fixes X Y
defines $Z \equiv X \cup \{x.\ x \in Y \wedge set\ x \cap \bigcup(set\ `\ X) = \{\}\}$
assumes $\forall\, xs \in Y.\ \forall\, ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
and $\forall\, xs \in X.\ \forall\, ys \in X.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
shows $\forall\, xs \in Z.\ \forall\, ys \in Z.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
⟨*proof*⟩

lemma *combine-union-sets-set-sub1-aux*:
assumes $\forall\, xs \in Y.\ \forall\, ys \in Y.\ xs = ys \vee set\ xs \cap set\ ys = \{\}$
and $\forall\, ys \in X.\ \exists\, U \in Y.\ \exists\, V \in Y.\ U@V = ys$
and $x \in \bigcup(set\ `\ Y)$
shows $x \in \bigcup(set\ `\ (X \cup \{x.\ x \in Y \wedge set\ x \cap \bigcup(set\ `\ X) = \{\}\}))$
⟨*proof*⟩

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 $\bigcup(set\ `\ Y) \subseteq \bigcup(set\ `\ (X \cup \{x.\ x \in Y \wedge set\ x \cap \bigcup(set\ `\ X) = \{\}\}))$
⟨*proof*⟩

lemma *combine-union-sets-set-sub2*:
assumes $\forall\, ys \in X.\ \exists\, U \in Y.\ \exists\, V \in Y.\ U@V = ys$
shows $\bigcup(set\ `\ (X \cup \{x.\ x \in Y \wedge set\ x \cap \bigcup(set\ `\ X) = \{\}\})) \subseteq \bigcup(set\ `\ Y)$
⟨*proof*⟩

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 $\bigcup(set\ `\ (X \cup \{x.\ x \in Y \wedge set\ x \cap \bigcup(set\ `\ X) = \{\}\})) = \bigcup(set\ `\ Y)$
⟨*proof*⟩

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*
⟨*proof*⟩

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 $\exists\, x.\ fwd\text{-}sub\ root\ Y\ x$
and $\forall\, ys \in X.\ \exists\, U \in Y.\ \exists\, V \in Y.\ U @ V = ys \wedge before\ U\ V \wedge rank\ (rev\ V) \leq rank\ (rev\ U)$
$\wedge\ (\forall\, xs \in Y.\ (\exists\, y{\in}set\ xs.\ \neg(\exists\, x'{\in}set\ V.\ x' \rightarrow^+_T y) \wedge (\exists\, x{\in}set\ U.\ x \rightarrow^+_T y) \wedge xs \neq U)$
$\longrightarrow rank\ (rev\ V) \leq rank\ (rev\ xs))$
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 $\exists\, zs.\ fwd\text{-}sub\ root\ (X \cup \{x.\ x \in Y \wedge set\ x \cap \bigcup(set\ `\ X) = \{\}\})\ zs$
$\wedge\ (\forall\, as.\ fwd\text{-}sub\ root\ Y\ as \longrightarrow cost\ (rev\ zs) \leq cost\ (rev\ as))$
⟨*proof*⟩

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*⟩

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*⟩

lemma *app-UV-set-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 $U \in Y$
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 $\exists zs.\ fwd\text{-}sub\ root\ (\{U@V\} \cup \{x.\ x \in Y \wedge x \neq U \wedge x \neq V\})\ zs$
$\wedge\ (\forall as.\ fwd\text{-}sub\ root\ Y\ as \longrightarrow cost\ (rev\ zs) \leq cost\ (rev\ as))$
⟨*proof*⟩

end

context *tree-query-graph*
begin

lemma *no-cross-ldeep-rev-if-forward*:
assumes $xs \neq []$ **and** $r \in verts\ G$ **and** *directed-tree.forward* (*dir-tree-r r*) (*rev xs*)
shows *no-cross-products* (*create-ldeep-rev xs*)
⟨*proof*⟩

lemma *no-cross-ldeep-if-forward*:
$[\![xs \neq []; r \in verts\ G; directed\text{-}tree.forward\ (dir\text{-}tree\text{-}r\ r)\ xs]\!]$
$\Longrightarrow$ *no-cross-products* (*create-ldeep xs*)
⟨*proof*⟩

lemma *no-cross-ldeep-if-forward'*:
$[\![set\ xs = verts\ G; r \in verts\ G; directed\text{-}tree.forward\ (dir\text{-}tree\text{-}r\ r)\ xs]\!]$
$\Longrightarrow$ *no-cross-products* (*create-ldeep xs*)
⟨*proof*⟩

lemma *forward-if-ldeep-rev-no-cross*:
assumes $r \in verts\ G$ **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*
⟨*proof*⟩

lemma *forward-if-ldeep-no-cross*:
$[\![r \in verts\ G; no\text{-}cross\text{-}products\ (create\text{-}ldeep\ xs); hd\ xs = r; distinct\ xs]\!]$
$\Longrightarrow$ *directed-tree.forward* (*dir-tree-r r*) *xs*
⟨*proof*⟩

lemma *no-cross-ldeep-iff-forward*:
$[\![xs \neq []; r \in verts\ G; hd\ xs = r; distinct\ xs]\!]$
$\Longrightarrow$ *no-cross-products* (*create-ldeep xs*) $\longleftrightarrow$ *directed-tree.forward* (*dir-tree-r r*)

xs
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*}
⟨*proof*⟩

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*)}
⟨*proof*⟩

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*
⟨*proof*⟩

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*)}⟧

$\Longrightarrow \exists\, y \in ys.\ \forall\, z \in ys.\ (f :: 'a\ list \Rightarrow real)\ y \leq f\ z$
⟨*proof*⟩

lemma *nocross-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 $Q = (\lambda ys.\ P\ ys \wedge no\text{-}cross\text{-}products\ (create\text{-}ldeep\ ys) \wedge (\forall\, xs \in Y.\ sublist\ xs\ ys))$
and $\exists\, x.\ Q\ x$
shows $\exists\, zs.\ Q\ zs \wedge (\forall\, as.\ Q\ as \longrightarrow f\ zs \leq f\ as)$
⟨*proof*⟩

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-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*⟩

end

10.3 Arc Invariants

function *path-lverts* :: $('a\ list,'b)\ dtree \Rightarrow 'a \Rightarrow 'a\ set$ **where**
path-lverts $(Node\ r\ \{|(t,e)|\})\ x = (if\ x \in set\ r\ then\ \{\}\ else\ set\ r \cup path\text{-}lverts\ t\ x)$
$|\ \forall\, x.\ xs \neq \{|x|\} \Longrightarrow path\text{-}lverts\ (Node\ r\ xs)\ x = (if\ x \in set\ r\ then\ \{\}\ else\ set\ r)$
⟨*proof*⟩
termination ⟨*proof*⟩

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* ⇒ (*'a*,*'b*) *pre-digraph* ⇒ *bool* **where**
dom-children t1 T = (∀ *t* ∈ *fst* ‘ *fset* (*sucs t1*). ∀ *x* ∈ *dverts t*.
∃ *r* ∈ *set* (*root t1*) ∪ *path-lverts t* (*hd x*). *r* →$_T$ *hd x*)

abbreviation *children-deg1* :: ((*'a*,*'b*) *dtree* × *'b*) *fset* ⇒ ((*'a*,*'b*) *dtree* × *'b*) *set* **where**
children-deg1 xs ≡ {(*t*,*e*). (*t*,*e*) ∈ *fset xs* ∧ *max-deg t* ≤ *1*}

lemma *path-lverts-subset-dlverts*: *path-lverts t x* ⊆ *dlverts t*
⟨*proof*⟩

lemma *path-lverts-to-list-eq*:
path-lverts t x = *path-lverts-list* (*dtree-to-list* (*Node r0* {|(*t*,*e*)|})) *x*
⟨*proof*⟩

lemma *path-lverts-from-list-eq*:
path-lverts (*dtree-from-list r0 ys*) *x* = *path-lverts-list* ((*r0*,*e0*)#*ys*) *x*
⟨*proof*⟩

lemma *path-lverts-child-union-root-sub*:
assumes *t2* ∈ *fst* ‘ *fset* (*sucs t1*)
shows *path-lverts t1 x* ⊆ *set* (*root t1*) ∪ *path-lverts t2 x*
⟨*proof*⟩

lemma *path-lverts-simps1-sucs*:
⟦*x* ∉ *set* (*root t1*); *sucs t1* = {|(*t2*,*e2*)|}⟧
⟹ *set* (*root t1*) ∪ *path-lverts t2 x* = *path-lverts t1 x*
⟨*proof*⟩

lemma *subtree-path-lverts-sub*:
⟦*wf-dlverts t1*; *max-deg t1* ≤ *1*; *is-subtree* (*Node r xs*) *t1*; *t2* ∈ *fst* ‘ *fset xs*; *x*∈*set* (*root t2*)⟧
⟹ *set r* ⊆ *path-lverts t1 x*
⟨*proof*⟩

lemma *path-lverts-empty-if-roothd*:
assumes *root t* ≠ []
shows *path-lverts t* (*hd* (*root t*)) = {}
⟨*proof*⟩

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*⟩

lemma *path-lverts-list-merge-supset-xs-notin*:
∀ *v* ∈ *fst* ‘ *set ys*. *a* ∉ *set v*
⟹ *path-lverts-list xs a* ⊆ *path-lverts-list* (*Sorting-Algorithms.merge cmp xs ys*) *a*

⟨*proof*⟩

lemma *path-lverts-list-merge-supset-ys-notin*:
∀ *v* ∈ *fst* ‘ *set xs*. *a* ∉ *set v*
⟹ *path-lverts-list ys a* ⊆ *path-lverts-list* (*Sorting-Algorithms.merge cmp xs ys*) *a*
⟨*proof*⟩

lemma *path-lverts-list-merge-supset-xs*:
⟦∃ *v* ∈ *fst* ‘ *set xs*. *a* ∈ *set v*; ∀ *v1* ∈ *fst* ‘ *set xs*. ∀ *v2* ∈ *fst* ‘ *set ys*. *set v1* ∩ *set v2* = {}⟧
⟹ *path-lverts-list xs a* ⊆ *path-lverts-list* (*Sorting-Algorithms.merge cmp xs ys*) *a*
⟨*proof*⟩

lemma *path-lverts-list-merge-supset-ys*:
⟦∃ *v* ∈ *fst* ‘ *set ys*. *a* ∈ *set v*; ∀ *v1* ∈ *fst* ‘ *set xs*. ∀ *v2* ∈ *fst* ‘ *set ys*. *set v1* ∩ *set v2* = {}⟧
⟹ *path-lverts-list ys a* ⊆ *path-lverts-list* (*Sorting-Algorithms.merge cmp xs ys*) *a*
⟨*proof*⟩

lemma *dom-children-if-all-singletons*:
∀ (*t1*,*e1*) ∈ *fset xs*. *dom-children* (*Node r* {|(*t1*, *e1*)|}) *T* ⟹ *dom-children* (*Node r xs*) *T*
⟨*proof*⟩

lemma *dom-children-all-singletons*:
⟦*dom-children* (*Node r xs*) *T*; (*t1*,*e1*) ∈ *fset xs*⟧ ⟹ *dom-children* (*Node r* {|(*t1*, *e1*)|}) *T*
⟨*proof*⟩

lemma *dom-children-all-singletons′*:
⟦*dom-children* (*Node r xs*) *T*; *t1*∈ *fst* ‘ *fset xs*⟧ ⟹ *dom-children* (*Node r* {|(*t1*, *e1*)|}) *T*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *root-arc-if-dom-wfdlverts*:
⟦*dom-children* (*Node r xs*) *T*; *t1* ∈ *fst* ‘ *fset xs*; *wf-dlverts* (*Node r xs*)⟧

$\Longrightarrow \exists\, x{\in}set\ r.\ \exists\, y{\in}set\ (root\ t1).\ x \rightarrow_T y$
⟨*proof*⟩

lemma *children-deg1-sub-xs*: $\{(t,e).\ (t,e) \in fset\ xs \wedge max\text{-}deg\ t \leq 1\} \subseteq (fset\ xs)$
⟨*proof*⟩

lemma *finite-children-deg1*: *finite* $\{(t,e).\ (t,e) \in fset\ xs \wedge max\text{-}deg\ t \leq 1\}$
⟨*proof*⟩

lemma *finite-children-deg1'*: $\{(t,e).\ (t,e) \in fset\ xs \wedge max\text{-}deg\ t \leq 1\} \in \{A.\ finite\ A\}$
⟨*proof*⟩

lemma *children-deg1-fset-id*[*simp*]: *fset* (*Abs-fset* (*children-deg1 xs*)) = *children-deg1 xs*
⟨*proof*⟩

lemma *xs-sub-children-deg1*: $\forall\, t \in fst\ `\ fset\ xs.\ max\text{-}deg\ t \leq 1 \Longrightarrow (fset\ xs) \subseteq children\text{-}deg1\ xs$
⟨*proof*⟩

lemma *children-deg1-full*:
$\forall\, t \in fst\ `\ fset\ xs.\ max\text{-}deg\ t \leq 1 \Longrightarrow (Abs\text{-}fset\ (children\text{-}deg1\ xs)) = xs$
⟨*proof*⟩

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*:
$[\![is\text{-}subtree\ (Node\ r\ xs)\ t;\ t1 \in fst\ `\ fset\ xs;\ max\text{-}deg\ (Node\ r\ xs) > 1]\!]$
$\Longrightarrow \exists\, v \in set\ r.\ v \rightarrow_T hd\ (Dtree.root\ t1)$
and *dom-sub-contr*:
$[\![is\text{-}subtree\ (Node\ r\ xs)\ t;\ t1 \in fst\ `\ fset\ xs;$
$\exists\, v\ t2\ e2.\ is\text{-}subtree\ (Node\ v\ \{|(t2,e2)|\})\ (Node\ r\ xs) \wedge rank\ (rev\ (Dtree.root\ t2)) < rank\ (rev\ v)]\!]$
$\Longrightarrow \exists\, v \in set\ r.\ v \rightarrow_T hd\ (Dtree.root\ t1)$
and *dom-contr*:
$[\![is\text{-}subtree\ (Node\ r\ \{|(t1,e1)|\})\ t;\ rank\ (rev\ (Dtree.root\ t1)) < rank\ (rev\ r);$
$max\text{-}deg\ (Node\ r\ \{|(t1,e1)|\}) = 1]\!]$
$\Longrightarrow dom\text{-}children\ (Node\ r\ \{|(t1,e1)|\})\ T$
and *dom-wedge*:
$[\![is\text{-}subtree\ (Node\ r\ xs)\ t;\ fcard\ xs > 1]\!]$
$\Longrightarrow dom\text{-}children\ (Node\ r\ (Abs\text{-}fset\ (children\text{-}deg1\ xs)))\ T$
and *arc-in-dlverts*:
$[\![is\text{-}subtree\ (Node\ r\ xs)\ t;\ x \in set\ r;\ x \rightarrow_T y]\!] \Longrightarrow y \in dlverts\ (Node\ r\ xs)$
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'*:
⟦*is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t*; *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*);
max-deg (*Node r* {|(*t1*,*e1*)|}) ≤ *1*⟧
⟹ *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T*
⟨*proof*⟩

lemma *dom-self-contr*:
⟦*is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t*; *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*)⟧
⟹ ∃ *v* ∈ *set r*. *v* →$_T$ *hd* (*Dtree.root t1*)
⟨*proof*⟩

lemma *dom-wedge-full*:
⟦*is-subtree* (*Node r xs*) *t*; *fcard xs* > *1*; ∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*⟧
⟹ *dom-children* (*Node r xs*) *T*
⟨*proof*⟩

lemma *dom-wedge-singleton*:
⟦*is-subtree* (*Node r xs*) *t*; *fcard xs* > *1*; *t1* ∈ *fst* ‘ *fset xs*; *max-deg t1* ≤ *1*⟧
⟹ *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T*
⟨*proof*⟩

lemma *arc-to-dverts-in-subtree*:
⟦*is-subtree* (*Node r xs*) *t*; *x* ∈ *set r*; *x* →$_T$ *y*; *y* ∈ *set v*; *v* ∈ *dverts t*⟧
⟹ *v* ∈ *dverts* (*Node r xs*)
⟨*proof*⟩

lemma *dlverts-arc-in-dlverts*:
⟦*is-subtree t1 t*; *x* →$_T$ *y*; *x* ∈ *dlverts t1*⟧ ⟹ *y* ∈ *dlverts t1*
⟨*proof*⟩

lemma *dverts-arc-in-dlverts*:
⟦*is-subtree t1 t*; *v1* ∈ *dverts t1*; *x* ∈ *set v1*; *x* →$_T$ *y*⟧ ⟹ *y* ∈ *dlverts t1*
⟨*proof*⟩

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*⟩

lemma *dlverts-reach1-in-dlverts*:
⟦*x* →$^+{}_T$ *y*; *is-subtree t1 t*; *x* ∈ *dlverts t1*⟧ ⟹ *y* ∈ *dlverts t1*
⟨*proof*⟩

lemma *dlverts-reach-in-dlverts*:
⟦$x \rightarrow^*{}_T\ y$; *is-subtree t1 t*; $x \in$ *dlverts t1*⟧ ⟹ $y \in$ *dlverts t1*
⟨*proof*⟩

lemma *dverts-reach1-in-dlverts*:
⟦*is-subtree t1 t*; *v1* $\in$ *dverts t1*; $x \in$ *set v1*; $x \rightarrow^+{}_T\ y$⟧ ⟹ $y \in$ *dlverts t1*
⟨*proof*⟩

lemma *dverts-reach-in-dlverts*:
⟦*is-subtree t1 t*; *v1* $\in$ *dverts t1*; $x \in$ *set v1*; $x \rightarrow^*{}_T\ y$⟧ ⟹ $y \in$ *dlverts t1*
⟨*proof*⟩

lemma *dverts-reach1-in-dverts*:
⟦*is-subtree t1 t*; *v1* $\in$ *dverts t1*; $x \in$ *set v1*; $x \rightarrow^+{}_T\ y$; $y \in$ *set v2*; *v2* $\in$ *dverts t*⟧
⟹ *v2* $\in$ *dverts t1*
⟨*proof*⟩

lemma *dverts-same-if-set-subtree*:
⟦*is-subtree t1 t*; *v1* $\in$ *dverts t1*; $x \in$ *set v1*; $x \in$ *set v2*; *v2* $\in$ *dverts t*⟧ ⟹ *v1* = *v2*
⟨*proof*⟩

lemma *dverts-reach-in-dverts*:
⟦*is-subtree t1 t*; *v1* $\in$ *dverts t1*; $x \in$ *set v1*; $x \rightarrow^*{}_T\ y$; $y \in$ *set v2*; *v2* $\in$ *dverts t*⟧
⟹ *v2* $\in$ *dverts t1*
⟨*proof*⟩

lemma *dverts-reach1-in-dverts-root*:
⟦*is-subtree t1 t*; *v* $\in$ *dverts t*; $\exists\, x{\in}$*set* (*Dtree.root t1*). $\exists\, y{\in}$*set v*. $x \rightarrow^+{}_T\ y$⟧
⟹ *v* $\in$ *dverts t1*
⟨*proof*⟩

lemma *dverts-reach1-in-dverts-r*:
⟦*is-subtree* (*Node r xs*) *t*; *v* $\in$ *dverts t*; $\exists\, x{\in}$*set r*. $\exists\, y{\in}$*set v*. $x \rightarrow^+{}_T\ y$⟧
⟹ *v* $\in$ *dverts* (*Node r xs*)
⟨*proof*⟩

lemma *dom-mdeg-gt1-subtree*:
⟦*is-subtree tn t*; *is-subtree* (*Node r xs*) *tn*; *t1* $\in$ *fst* ‘ *fset xs*; *max-deg* (*Node r xs*) > *1*⟧
⟹ $\exists\, v \in$ *set r*. $v \rightarrow_T$ *hd* (*Dtree.root t1*)
⟨*proof*⟩

lemma *dom-sub-contr-subtree*:
⟦*is-subtree tn t*; *is-subtree* (*Node r xs*) *tn*; *t1* $\in$ *fst* ‘ *fset xs*;
$\exists\, v$ *t2 e2*. *is-subtree* (*Node v* {|(*t2*,*e2*)|}) (*Node r xs*) $\wedge$ *rank* (*rev* (*Dtree.root t2*)) < *rank* (*rev v*)⟧
⟹ $\exists\, v \in$ *set r*. $v \rightarrow_T$ *hd* (*Dtree.root t1*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))))
⟨*proof*⟩

lemma *verts-conform-subtree*: ⟦*is-subtree tn t*; *v* ∈ *dverts tn*⟧ ⟹ *seq-conform v*
⟨*proof*⟩

lemma *verts-distinct-subtree*: ⟦*is-subtree tn t*; *v* ∈ *dverts tn*⟧ ⟹ *distinct v*
⟨*proof*⟩

lemma *ranked-dtree-orig-subtree*: *is-subtree x t* ⟹ *ranked-dtree-with-orig x rank cost cmp T root*
⟨*proof*⟩

corollary *ranked-dtree-orig-rec*:
⟦*Node r xs* = *t*; (*x*,*e*) ∈ *fset xs*⟧ ⟹ *ranked-dtree-with-orig x rank cost cmp T root*
⟨*proof*⟩

lemma *child-disjoint-root*:
⟦*is-subtree* (*Node r xs*) *t*; *t1* ∈ *fst* ‘ *fset xs*⟧ ⟹ *set r* ∩ *set* (*Dtree.root t1*) = {}
⟨*proof*⟩

lemma *distint-verts-subtree*:
assumes *is-subtree* (*Node r xs*) *t* **and** *t1* ∈ *fst* ‘ *fset xs*
shows *distinct* (*r* @ *Dtree.root t1*)
⟨*proof*⟩

corollary *distint-verts-singleton-subtree*:
is-subtree (*Node r* {|(*t1*,*e1*)|}) *t* ⟹ *distinct* (*r* @ *Dtree.root t1*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *verts-forward*: ∀ *v* ∈ *dverts t*. *forward v*
⟨*proof*⟩

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*⟩

lemma *dverts-reachable1-if-dom-children-aux*:
⟦∀ *v*∈*dverts t1*. ∃ *x*∈*set r0* ∪ *X* ∪ *path-lverts t1* (*hd v*). *x* →$_T$ *hd v*;

$\forall y{\in}X.\ \exists x{\in}set\ r0.\ x \to^+_T y;\ \forall v{\in}dverts\ t1.\ forward\ v;\ v{\in}dverts\ t1\rrbracket$
$\Longrightarrow \forall y{\in}set\ v.\ \exists x{\in}set\ r0.\ x \to^+_T y$
⟨*proof*⟩

lemma *dlverts-reachable1-if-dom-children-aux*:
$\llbracket\forall v{\in}dverts\ t1.\ \exists x{\in}set\ r \cup X \cup path\text{-}lverts\ t1\ (hd\ v).\ x \to_T hd\ v;$
$\forall y{\in}X.\ \exists x{\in}set\ r.\ x \to^+_T y;\ \forall v{\in}dverts\ t1.\ forward\ v;\ y{\in}dlverts\ t1\rrbracket$
$\Longrightarrow \exists x{\in}set\ r.\ x \to^+_T y$
⟨*proof*⟩

lemma *dverts-reachable1-if-dom-children*:
assumes *dom-children t1 T* **and** $v \in dverts\ t1$ **and** $v \neq Dtree.root\ t1$ **and** $\forall v{\in}dverts\ t1.\ forward\ v$
shows $\forall y{\in}set\ v.\ \exists x{\in}set\ (Dtree.root\ t1).\ x \to^+_T y$
⟨*proof*⟩

lemma *subtree-dverts-reachable1-if-mdeg-gt1*:
$\llbracket is\text{-}subtree\ t1\ t;\ max\text{-}deg\ t1 > 1;\ v \in dverts\ t1;\ v \neq Dtree.root\ t1\rrbracket$
$\Longrightarrow \forall y{\in}set\ v.\ \exists x{\in}set\ (Dtree.root\ t1).\ x \to^+_T y$
⟨*proof*⟩

lemma *subtree-dverts-reachable1-if-mdeg-gt1-singleton*:
assumes $is\text{-}subtree\ (Node\ r\ \{|(t1,e1)|\})\ t$
and $max\text{-}deg\ (Node\ r\ \{|(t1,e1)|\}) > 1$
and $v \in dverts\ t1$
and $v \neq Dtree.root\ t1$
shows $\forall y{\in}set\ v.\ \exists x{\in}set\ (Dtree.root\ t1).\ x \to^+_T y$
⟨*proof*⟩

lemma *subtree-dverts-reachable1-if-mdeg-le1-subcontr*:
$\llbracket is\text{-}subtree\ t1\ t;\ max\text{-}deg\ t1 \leq 1;\ is\text{-}subtree\ (Node\ v2\ \{|(t2,e2)|\})\ t1;$
$rank\ (rev\ (Dtree.root\ t2)) < rank\ (rev\ v2);\ v \in dverts\ t1;\ v \neq Dtree.root\ t1\rrbracket$
$\Longrightarrow \forall y{\in}set\ v.\ \exists x{\in}set\ (Dtree.root\ t1).\ x \to^+_T y$
⟨*proof*⟩

lemma *subtree-y-reach-if-mdeg-gt1-notroot-reach*:
assumes $is\text{-}subtree\ (Node\ r\ \{|(t1,e1)|\})\ t$
and $max\text{-}deg\ (Node\ r\ \{|(t1,e1)|\}) > 1$
and $v \neq r$
and $v \in dverts\ t$
and $v \neq Dtree.root\ t1$
and $y \in set\ v$
and $\exists x{\in}set\ r.\ x \to^+_T y$
shows $\exists x'{\in}set\ (Dtree.root\ t1).\ x' \to^+_T y$
⟨*proof*⟩

lemma *subtree-eqroot-if-mdeg-gt1-reach*:
$\llbracket is\text{-}subtree\ (Node\ r\ \{|(t1,e1)|\})\ t;\ max\text{-}deg\ (Node\ r\ \{|(t1,e1)|\}) > 1;\ v \in dverts\ t;$

$\exists\, y{\in}set\ v.\ \neg(\exists\, x'{\in}set\ (Dtree.root\ t1).\ x' \to^+_T y) \wedge (\exists\, x{\in}set\ r.\ x \to^+_T y);\ v \neq r\rrbracket$
$\Longrightarrow Dtree.root\ t1 = v$
⟨*proof*⟩

lemma *subtree-rank-ge-if-mdeg-gt1-reach*:
$\llbracket$*is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t*; *max-deg* (*Node r* {|(*t1*,*e1*)|}) > *1*; *v* ∈ *dverts t*;
$\exists\, y{\in}set\ v.\ \neg(\exists\, x'{\in}set\ (Dtree.root\ t1).\ x' \to^+_T y) \wedge (\exists\, x{\in}set\ r.\ x \to^+_T y);\ v \neq r\rrbracket$
$\Longrightarrow$ *rank* (*rev* (*Dtree.root t1*)) ≤ *rank* (*rev v*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *subtree-rank-ge-if-mdeg-le1'*:
$\llbracket$*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

$y)[\![$
$\Longrightarrow$ *rank* (*rev* (*Dtree.root t1*)) $\leq$ *rank* (*rev v*)
⟨*proof*⟩

lemma *subtree-rank-ge-if-mdeg-le1*:
$[\![$*is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t*; *max-deg* (*Node r* {|(*t1*,*e1*)|}) $\leq$ *1*; $v \neq r$;
$v \in$ *dverts t*; $\exists\, y \in$ *set v*. $\neg(\exists\, x' \in$*set* (*Dtree.root t1*). $x' \rightarrow^+_T y) \wedge (\exists\, x \in$*set r*. $x \rightarrow^+_T y)[\![$
$\Longrightarrow$ *rank* (*rev* (*Dtree.root t1*)) $\leq$ *rank* (*rev v*)
⟨*proof*⟩

lemma *subtree-rank-ge-if-reach*:
$[\![$*is-subtree* (*Node r* {|(*t1*,*e1*)|}) *t*; $v \neq r$; $v \in$ *dverts t*;
$\exists\, y \in$ *set v*. $\neg(\exists\, x' \in$*set* (*Dtree.root t1*). $x' \rightarrow^+_T y) \wedge (\exists\, x \in$*set r*. $x \rightarrow^+_T y)[\![$
$\Longrightarrow$ *rank* (*rev* (*Dtree.root t1*)) $\leq$ *rank* (*rev v*)
⟨*proof*⟩

lemma *subtree-rank-ge-if-reach′*:
is-subtree (*Node r* {|(*t1*,*e1*)|}) *t* $\Longrightarrow \forall\, v \in$ *dverts t*.
$(\exists\, y \in$*set v*. $\neg$ $(\exists\, x' \in$*set* (*Dtree.root t1*). $x' \rightarrow^+_T y) \wedge (\exists\, x \in$*set r*. $x \rightarrow^+_T y) \wedge$
$v \neq r$)
$\longrightarrow$ *rank* (*rev* (*Dtree.root t1*)) $\leq$ *rank* (*rev v*)
⟨*proof*⟩

10.3.1 Normalizing preserves Arc Invariants

lemma *normalize1-mdeg-le*: *max-deg* (*normalize1 t1*) $\leq$ *max-deg t1*
⟨*proof*⟩

lemma *normalize1-mdeg-eq*:
wf-darcs t1
$\Longrightarrow$ *max-deg* (*normalize1 t1*) = *max-deg t1* $\vee$ (*max-deg* (*normalize1 t1*) = *0* $\wedge$
max-deg t1 = *1*)
⟨*proof*⟩

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*⟩

lemma *normalize1-dom-mdeg-gt1*:
$[\![$*is-subtree* (*Node r xs*) (*normalize1 t*); *t1* $\in$ *fst* ‘ *fset xs*; *max-deg* (*Node r xs*) $>$
1$]\!]$
$\Longrightarrow \exists\, v \in$ *set r*. $v \rightarrow_T$ *hd* (*Dtree.root t1*)
⟨*proof*⟩

lemma *child-contr-if-new-contr*:
assumes $\neg$*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*⟩

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*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *dom-children-combine*:
dom-children (*Node r* {|(*t1*, *e1*)|}) *T* ⟹ *dom-children* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*)) *T*
⟨*proof*⟩

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*⟩

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 $x \in$ *dverts (normalize1 t1)*
shows $\exists\, v \in$ *set r* $\cup$ *path-lverts (normalize1 t1) (hd x).* $v \rightarrow_T$ *(hd x)*
⟨*proof*⟩

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*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *dom-children-normalize1*:
⟦*dom-children (Node r0 {|(t1,e1)|}) T*; *wf-dlverts t1*; *max-deg t1* $\leq$ *1*⟧
$\Longrightarrow$ *dom-children (Node r0 {|(normalize1 t1,e1)|}) T*
⟨*proof*⟩

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 $x \in$ *dverts t1*
shows $\exists\, v \in$ *set r* $\cup$ *path-lverts t1 (hd x).* $v \rightarrow_T$ *hd x*
⟨*proof*⟩

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*
⟨*proof*⟩

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⟧
$\Longrightarrow$ *dom-children (Node r {|(t1,e1)|}) T*
⟨*proof*⟩

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*⟩

lemma *children-deg1-normalize1-sub*:
(λ(*t1*,*e1*). (*normalize1 t1*,*e1*)) ‘ *children-deg1 xs*
⊆ *children-deg1* ((λ(*t1*,*e1*). (*normalize1 t1*,*e1*)) |‘| *xs*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *normalize1-dom-wedge*:
⟦*is-subtree* (*Node r xs*) (*normalize1 t*); *fcard xs* > *1*⟧
⟹ *dom-children* (*Node r* (*Abs-fset* (*children-deg1 xs*))) *T*
⟨*proof*⟩

corollary *normalize1-dom-wedge′*:
$\forall r\ xs.\ is\text{-}subtree\ (Node\ r\ xs)\ (normalize1\ t) \longrightarrow fcard\ xs > 1$
$\longrightarrow dom\text{-}children\ (Node\ r\ (Abs\text{-}fset\ \{(t,\ e).\ (t,\ e) \in fset\ xs \wedge max\text{-}deg\ t \leq Suc\ 0\}))\ T$
⟨*proof*⟩

lemma *normalize1-verts-conform*: $v \in dverts\ (normalize1\ t) \Longrightarrow seq\text{-}conform\ v$
⟨*proof*⟩

corollary *normalize1-verts-distinct*: $v \in dverts\ (normalize1\ t) \Longrightarrow distinct\ v$
⟨*proof*⟩

lemma *dom-mdeg-le1-aux*:
assumes $max\text{-}deg\ t \leq 1$
and $is\text{-}subtree\ (Node\ v\ \{|(t2,\ e2)|\})\ t$
and $rank\ (rev\ (Dtree.root\ t2)) < rank\ (rev\ v)$
and $t1 \in fst\ `\ fset\ (sucs\ t)$
and $x \in dverts\ t1$
shows $\exists r{\in}set\ (Dtree.root\ t) \cup path\text{-}lverts\ t1\ (hd\ x).\ r \rightarrow_T hd\ x$
⟨*proof*⟩

lemma *dom-mdeg-le1*:
assumes $max\text{-}deg\ t \leq 1$
and $is\text{-}subtree\ (Node\ v\ \{|(t2,\ e2)|\})\ t$
and $rank\ (rev\ (Dtree.root\ t2)) < rank\ (rev\ v)$
shows $dom\text{-}children\ t\ T$
⟨*proof*⟩

lemma *dom-children-normalize1-preserv*:
assumes $max\text{-}deg\ (normalize1\ t1) \leq 1$ **and** $dom\text{-}children\ t1\ T$ **and** $wf\text{-}dlverts\ t1$
shows $dom\text{-}children\ (normalize1\ t1)\ T$
⟨*proof*⟩

lemma *dom-mdeg-le1-normalize1*:
assumes $max\text{-}deg\ (normalize1\ t) \leq 1$ **and** $normalize1\ t \neq t$
shows $dom\text{-}children\ (normalize1\ t)\ T$
⟨*proof*⟩

lemma *normalize-mdeg-eq*:
$wf\text{-}darcs\ t1$
$\Longrightarrow max\text{-}deg\ (normalize\ t1) = max\text{-}deg\ t1 \vee (max\text{-}deg\ (normalize\ t1) = 0 \wedge max\text{-}deg\ t1 = 1)$
⟨*proof*⟩

lemma *normalize-mdeg-eq′*:
$wf\text{-}dlverts\ t1$
$\Longrightarrow max\text{-}deg\ (normalize\ t1) = max\text{-}deg\ t1 \vee (max\text{-}deg\ (normalize\ t1) = 0 \wedge max\text{-}deg\ t1 = 1)$

⟨*proof*⟩

corollary *mdeg-le1-normalize*:
⟦*max-deg* (*normalize t1*) ≤ *1*; *wf-dlverts t1*⟧ ⟹ *max-deg t1* ≤ *1*
⟨*proof*⟩

lemma *dom-children-normalize-preserv*:
assumes *max-deg* (*normalize t1*) ≤ *1* **and** *dom-children t1 T* **and** *wf-dlverts t1*
shows *dom-children* (*normalize t1*) *T*
⟨*proof*⟩

lemma *dom-mdeg-le1-normalize*:
assumes *max-deg* (*normalize t*) ≤ *1* **and** *normalize t* ≠ *t*
shows *dom-children* (*normalize t*) *T*
⟨*proof*⟩

lemma *normalize1-arc-in-dlverts*:
⟦*is-subtree* (*Node v ys*) (*normalize1 t*); *x* ∈ *set v*; *x* →$_T$ *y*⟧ ⟹ *y* ∈ *dlverts* (*Node v ys*)
⟨*proof*⟩

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*))))
⟨*proof*⟩

theorem *ranked-dtree-orig-normalize1*: *ranked-dtree-with-orig* (*normalize1 t*) *rank cost cmp T root*
⟨*proof*⟩

theorem *ranked-dtree-orig-normalize*: *ranked-dtree-with-orig* (*normalize t*) *rank cost cmp T root*
⟨*proof*⟩

10.3.2 Merging preserves Arc Invariants

interpretation *Comm*: *comp-fun-commute merge-f r xs* ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *merge-dom-children*:
⟦*dom-children* (*Node r xs*) *T*; ∀ *t1* ∈ *fst* ‘ *fset xs*. *wf-dlverts t1*⟧
⟹ *dom-children* (*merge* (*Node r xs*)) *T*
⟨*proof*⟩

lemma *merge-dom-children-if-ndisjoint*:
¬*list-dtree* (*Node r xs*) ⟹ *dom-children* (*merge* (*Node r xs*)) *T*
⟨*proof*⟩

lemma *merge-subtree-fcard-le1*: *is-subtree* (*Node r xs*) (*merge t1*) $\Longrightarrow$ *fcard xs* $\leq$ *1*
⟨*proof*⟩

lemma *merge-dom-mdeg-gt1*:
⟦*is-subtree* (*Node r xs*) (*merge t2*); *t1* $\in$ *fst* ‘ *fset xs*; *max-deg* (*Node r xs*) > *1*⟧
$\Longrightarrow \exists\, v \in$ *set r*. *v* $\rightarrow_T$ *hd* (*Dtree.root t1*)
⟨*proof*⟩

lemma *merge-root-if-contr*:
⟦$\bigwedge$*r1 t2 e2*. *is-subtree* (*Node r1* {|(*t2*,*e2*)|}) *t1* $\Longrightarrow$ *rank* (*rev r1*) $\leq$ *rank* (*rev* (*Dtree.root t2*));
is-subtree (*Node v* {|(*t2*,*e2*)|}) (*merge t1*); *rank* (*rev* (*Dtree.root t2*)) < *rank* (*rev v*)⟧
$\Longrightarrow$ *Node v* {|(*t2*,*e2*)|} = *merge t1*
⟨*proof*⟩

lemma *merge-new-contr-fcard-gt1*:
assumes $\bigwedge$*r1 t2 e2*. *is-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*⟩

lemma *merge-dom-sub-contr-if-nocontr*:
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 xs*) (*merge t*)
and *t1* $\in$ *fst* ‘ *fset xs*
and $\exists$ *v t2 e2*. *is-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*⟩

lemma *merge-dom-contr-if-nocontr-mdeg-le1*:
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*)|}) (*merge t*)
and *rank* (*rev* (*Dtree.root t1*)) < *rank* (*rev r*)
and $\forall\, t \in$ *fst* ‘ *fset* (*sucs t*). *max-deg t* $\leq$ *1*
shows *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T*
⟨*proof*⟩

lemma *merge-dom-wedge*:
⟦*is-subtree* (*Node r xs*) (*merge t1*); *fcard xs* > *1*; $\forall\, t \in$ *fst* ‘ *fset xs*. *max-deg t* $\leq$ *1*⟧
$\Longrightarrow$ *dom-children* (*Node r xs*) *T*

⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *merge1-dom-sub-contr*:
assumes ⋀*r1 t2 e2*. *is-subtree* (*Node r1* {|(*t2*,*e2*)|}) *t* ⟹ *rank* (*rev r1*) ≤ *rank* (*rev* (*Dtree.root t2*))
and *is-subtree* (*Node r xs*) (*merge1 t*)
and *t1* ∈ *fst* ‘ *fset xs*
and ∃ *v t2 e2*. *is-subtree* (*Node v* {|(*t2*,*e2*)|}) (*Node r xs*)∧*rank* (*rev* (*Dtree.root t2*))<*rank* (*rev v*)
shows ∃ *v* ∈ *set r*. *v* →$_T$ *hd* (*Dtree.root t1*)
⟨*proof*⟩

lemma *merge1-merge-point-if-new-contr*:
assumes ⋀*r1 t2 e2*. *is-subtree* (*Node r1* {|(*t2*,*e2*)|}) *t0* ⟹ *rank* (*rev r1*) ≤ *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 ∃ *ys*. *is-subtree* (*Node r ys*) *t0* ∧ *fcard ys* > *1* ∧ (∀ *t*∈ *fst* ‘ *fset ys*. *max-deg t* ≤ *1*)
∧ *merge1* (*Node r ys*) = *Node r* {|(*t1*,*e1*)|}
⟨*proof*⟩

lemma *merge1-dom-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 $is\text{-}subtree\ (Node\ r\ \{|(t1,e1)|\})\ (merge1\ t)$
and $rank\ (rev\ (Dtree.root\ t1)) < rank\ (rev\ r)$
and $max\text{-}deg\ (Node\ r\ \{|(t1,e1)|\}) = 1$
shows $dom\text{-}children\ (Node\ r\ \{|(t1,e1)|\})\ T$
⟨*proof*⟩

lemma *merge1-dom-children-merge-sub-aux*:
assumes $merge1\ t = t2$
and $is\text{-}subtree\ (Node\ r'\ xs')\ t$
and $fcard\ xs' > 1$
and $(\forall\, t{\in}fst\ `\ fset\ xs'.\ max\text{-}deg\ t \leq 1)$
and $max\text{-}deg\ t2 \leq 1$
and $x \in dverts\ t2$
and $x \neq Dtree.root\ t2$
shows $\exists\, v \in path\text{-}lverts\ t2\ (hd\ x).\ v \rightarrow_T hd\ x$
⟨*proof*⟩

lemma *merge1-dom-children-fcard-gt1-aux*:
assumes $dom\text{-}children\ (Node\ r\ (Abs\text{-}fset\ (children\text{-}deg1\ ys)))\ T$
and $is\text{-}subtree\ (Node\ r\ ys)\ t$
and $merge1\ (Node\ r\ ys) = Node\ r\ xs$
and $fcard\ xs > 1$
and $max\text{-}deg\ t2 \leq 1$
and $t2 \in fst\ `\ fset\ xs$
and $x \in dverts\ t2$
shows $\exists\, v{\in}set\ r \cup path\text{-}lverts\ t2\ (hd\ x).\ v \rightarrow_T hd\ x$
⟨*proof*⟩

lemma *merge1-dom-children-fcard-gt1*:
assumes $dom\text{-}children\ (Node\ r\ (Abs\text{-}fset\ (children\text{-}deg1\ ys)))\ T$
and $is\text{-}subtree\ (Node\ r\ ys)\ t$
and $merge1\ (Node\ r\ ys) = Node\ r\ xs$
and $fcard\ xs > 1$
shows $dom\text{-}children\ (Node\ r\ (Abs\text{-}fset\ (children\text{-}deg1\ xs)))\ T$
⟨*proof*⟩

lemma *merge1-dom-wedge*:
assumes $is\text{-}subtree\ (Node\ r\ xs)\ (merge1\ t)$ **and** $fcard\ xs > 1$
shows $dom\text{-}children\ (Node\ r\ (Abs\text{-}fset\ (children\text{-}deg1\ xs)))\ T$
⟨*proof*⟩

corollary *merge1-dom-wedge'*:
$\forall\, r\ xs.\ is\text{-}subtree\ (Node\ r\ xs)\ (merge1\ t) \longrightarrow fcard\ xs > 1$
$\longrightarrow dom\text{-}children\ (Node\ r\ (Abs\text{-}fset\ \{(t,\ e).\ (t,\ e) \in fset\ xs \wedge max\text{-}deg\ t \leq Suc\ 0\}))\ T$
⟨*proof*⟩

corollary *merge1-verts-conform*: $v \in dverts\ (merge1\ t) \Longrightarrow seq\text{-}conform\ v$
⟨*proof*⟩

corollary *merge1-verts-distinct*: $[\![v \in dverts\ (merge1\ t)]\!] \Longrightarrow distinct\ v$
⟨*proof*⟩

lemma *merge1-mdeg-le1-wedge-if-fcard-gt1*:
assumes $max\text{-}deg\ (merge1\ t1) \leq 1$
and *wf-darcs t1*
and *is-subtree* (*Node v ys*) *t1*
and $fcard\ ys > 1$
shows $(\forall t \in fst\ `\ fset\ ys.\ max\text{-}deg\ t \leq 1)$
⟨*proof*⟩

lemma *dom-mdeg-le1-merge1-aux*:
assumes $max\text{-}deg\ (merge1\ t) \leq 1$
and $merge1\ t \neq t$
and $t1 \in fst\ `\ fset\ (sucs\ (merge1\ t))$
and $x \in dverts\ t1$
shows $\exists r{\in}set\ (Dtree.root\ (merge1\ t)) \cup path\text{-}lverts\ t1\ (hd\ x).\ r \rightarrow_T hd\ x$
⟨*proof*⟩

lemma *dom-mdeg-le1-merge1*:
$[\![max\text{-}deg\ (merge1\ t) \leq 1;\ merge1\ t \neq t]\!] \Longrightarrow dom\text{-}children\ (merge1\ t)\ T$
⟨*proof*⟩

lemma *merge1-arc-in-dlverts*:
$[\![is\text{-}subtree\ (Node\ r\ xs)\ (merge1\ t);\ x \in set\ r;\ x \rightarrow_T y]\!] \Longrightarrow y \in dlverts\ (Node\ r\ xs)$
⟨*proof*⟩

theorem *merge1-ranked-dtree-orig*:
assumes $\bigwedge r1\ t2\ e2.\ is\text{-}subtree\ (Node\ r1\ \{|(t2,e2)|\})\ t \Longrightarrow rank\ (rev\ r1) \leq rank\ (rev\ (Dtree.root\ t2))$
shows *ranked-dtree-with-orig* (*merge1 t*) *rank cost cmp T root*
⟨*proof*⟩

theorem *merge1-normalize-ranked-dtree-orig*:
ranked-dtree-with-orig (*merge1* (*normalize t*)) *rank cost cmp T root*
⟨*proof*⟩

theorem *ikkbz-sub-ranked-dtree-orig*: *ranked-dtree-with-orig* (*ikkbz-sub t*) *rank cost cmp T root*
⟨*proof*⟩

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*)|})
⟨*proof*⟩

lemma *dtree-size-skip-decr1*: *size* (*Node* (*r* @ *Dtree.root t1*) (*sucs t1*)) < *size* (*Node r* {|(*t1*,*e1*)|})
⟨*proof*⟩

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*
⟨*proof*⟩
termination ⟨*proof*⟩

10.4.1 Result fulfills the requirements

lemma *ikkbz-sub-eq-if-mdeg-le1*: *max-deg t1* ≤ *1* ⟹ *ikkbz-sub t1* = *t1*
⟨*proof*⟩

lemma *ikkbz-sub-eq-iff-mdeg-le1*: *max-deg t1* ≤ *1* ⟷ *ikkbz-sub t1* = *t1*
⟨*proof*⟩

lemma *dom-mdeg-le1-ikkbz-sub*: *ikkbz-sub t* ≠ *t* ⟹ *dom-children* (*ikkbz-sub t*) *T*
⟨*proof*⟩

lemma *combine-denormalize-eq*:
denormalize (*Node r* {|(*t1*,*e1*)|}) = *denormalize* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
⟨*proof*⟩

lemma *normalize1-denormalize-eq*: *wf-dlverts t1* ⟹ *denormalize* (*normalize1 t1*) = *denormalize t1*
⟨*proof*⟩

lemma *normalize1-denormalize-eq'*: *wf-darcs t1* ⟹ *denormalize* (*normalize1 t1*) = *denormalize t1*
⟨*proof*⟩

lemma *normalize-denormalize-eq*: *wf-dlverts t1* ⟹ *denormalize* (*normalize t1*) = *denormalize t1*
⟨*proof*⟩

lemma *normalize-denormalize-eq'*: *wf-darcs t1* ⟹ *denormalize* (*normalize t1*) = *denormalize t1*
⟨*proof*⟩

lemma *normalize-full-denormalize-eq*[*simp*]: *denormalize* (*normalize-full t1*) = *denormalize t1*
⟨*proof*⟩

lemma *combine-dlverts-eq*: *dlverts* (*Node r* {|(*t1*,*e1*)|}) = *dlverts* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
⟨*proof*⟩

lemma *normalize-full-dlverts-eq*[*simp*]: *dlverts* (*normalize-full t1*) = *dlverts t1*
⟨*proof*⟩

lemma *combine-darcs-sub*: *darcs* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*)) ⊆ *darcs* (*Node r* {|(*t1*,*e1*)|})
⟨*proof*⟩

lemma *normalize-full-darcs-sub*: *darcs* (*normalize-full t1*) ⊆ *darcs t1*
⟨*proof*⟩

lemma *combine-nempty-if-wf-dlverts*: *wf-dlverts* (*Node r* {|(*t1*,*e1*)|}) ⟹ *r* @ *Dtree.root t1* ≠ []
⟨*proof*⟩

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*⟩

lemma *combine-disjoint-if-wf-dlverts*:
wf-dlverts (*Node r* {|(*t1*,*e1*)|}) ⟹ *disjoint-dlverts* (*sucs t1*)
⟨*proof*⟩

lemma *combine-wf-dlverts*:
wf-dlverts (*Node r* {|(*t1*,*e1*)|}) ⟹ *wf-dlverts* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
⟨*proof*⟩

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*⟩

lemma *normalize-full-wfdlverts*: *wf-dlverts t1* ⟹ *wf-dlverts* (*normalize-full t1*)
⟨*proof*⟩

corollary *normalize-full-wfdverts*: *wf-dlverts t1* ⟹ *wf-dverts* (*normalize-full t1*)
⟨*proof*⟩

lemma *combine-wf-arcs*: *wf-darcs* (*Node r* {|(*t1*,*e1*)|}) ⟹ *wf-darcs* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
⟨*proof*⟩

lemma *normalize-full-wfdarcs*: *wf-darcs t1* ⟹ *wf-darcs* (*normalize-full t1*)
⟨*proof*⟩

lemma *normalize-full-dom-preserv*: *dom-children t1 T* ⟹ *dom-children* (*normalize-full t1*) *T*
⟨*proof*⟩

lemma *combine-forward*:
assumes *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T*
and ∀ *v* ∈ *dverts* (*Node r* {|(*t1*,*e1*)|}). *forward v*
and *wf-dlverts* (*Node r* {|(*t1*,*e1*)|})
and *v* ∈ *dverts* (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
shows *forward v*
⟨*proof*⟩

lemma *normalize-full-forward*:
⟦*dom-children t1 T*; ∀ *v* ∈ *dverts t1*. *forward v*; *wf-dlverts t1*⟧
⟹ ∀ *v* ∈ *dverts* (*normalize-full t1*). *forward v*
⟨*proof*⟩

lemma *normalize-full-max-deg0*: *max-deg t1* ≤ *1* ⟹ *max-deg* (*normalize-full t1*) = *0*
⟨*proof*⟩

lemma *normalize-full-mdeg-eq*: *max-deg t1* > *1* ⟹ *max-deg* (*normalize-full t1*) = *max-deg t1*
⟨*proof*⟩

lemma *normalize-full-empty-sucs*: *max-deg t1* ≤ *1* ⟹ ∃ *r*. *normalize-full t1* = *Node r* {||}
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *denormalize-empty-sucs-simp*: *denormalize* (*Node r* {||}) = *r*
⟨*proof*⟩

lemma *normalize-full-dverts-eq-denormalize*:
assumes *max-deg t1* ≤ *1*
shows *dverts* (*normalize-full t1*) = {*denormalize t1*}
⟨*proof*⟩

lemma *normalize-full-normalize-dverts-eq-denormalize*:
assumes *wf-dlverts t1* **and** *max-deg t1* $\leq$ *1*
shows *dverts* (*normalize-full* (*normalize t1*)) = {*denormalize t1*}
⟨*proof*⟩

lemma *normalize-full-normalize-dverts-eq-denormalize'*:
assumes *wf-darcs t1* **and** *max-deg t1* $\leq$ *1*
shows *dverts* (*normalize-full* (*normalize t1*)) = {*denormalize t1*}
⟨*proof*⟩

lemma *denormalize-full-forward*:
⟦*max-deg t1* $\leq$ *1*; *dom-children t1 T*; $\forall$ *v* $\in$ *dverts t1*. *forward v*; *wf-dlverts t1*⟧
$\Longrightarrow$ *forward* (*denormalize* (*normalize-full t1*))
⟨*proof*⟩

lemma *denormalize-forward*:
⟦*max-deg t1* $\leq$ *1*; *dom-children t1 T*; $\forall$ *v* $\in$ *dverts t1*. *forward v*; *wf-dlverts t1*⟧
$\Longrightarrow$ *forward* (*denormalize t1*)
⟨*proof*⟩

lemma *ikkbz-sub-forward-if-uneq*: *ikkbz-sub t* $\neq$ *t* $\Longrightarrow$ *forward* (*denormalize* (*ikkbz-sub t*))
⟨*proof*⟩

theorem *ikkbz-sub-forward*:
⟦*max-deg t* $\leq$ *1* $\Longrightarrow$ *dom-children t T*⟧ $\Longrightarrow$ *forward* (*denormalize* (*ikkbz-sub t*))
⟨*proof*⟩

lemma *root-arc-singleton*:
assumes *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T* **and** *wf-dlverts* (*Node r* {|(*t1*,*e1*)|})
shows $\exists$ *x*$\in$*set r*. $\exists$ *y*$\in$*set* (*Dtree.root t1*). *x* $\rightarrow_T$ *y*
⟨*proof*⟩

lemma *before-if-dom-children-wf-conform*:
assumes *dom-children* (*Node r* {|(*t1*,*e1*)|}) *T*
and $\forall$ *v* $\in$ *dverts* (*Node r* {|(*t1*,*e1*)|}). *seq-conform v*
and *wf-dlverts* (*Node r* {|(*t1*,*e1*)|})
shows *before r* (*Dtree.root t1*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *root-before-if-dom*:
assumes *Node r* {|(*t1*,*e1*)|} = *t* **and** *dom-children t T*
shows *before r* (*Dtree.root t1*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *denormalize-full-set-eq-dlverts*:
max-deg t1 ≤ *1* ⟹ *set* (*denormalize* (*normalize-full t1*)) = *dlverts t1*
⟨*proof*⟩

lemma *denormalize-full-set-eq-dverts-union*:
max-deg t1 ≤ *1* ⟹ *set* (*denormalize* (*normalize-full t1*)) = ⋃(*set* ‘ *dverts t1*)
⟨*proof*⟩

corollary *hd-eq-denormalize-full*:
wf-dlverts t1 ⟹ *hd* (*denormalize* (*normalize-full t1*)) = *hd* (*Dtree.root t1*)
⟨*proof*⟩

corollary *denormalize-full-nempty-if-wf*:
wf-dlverts t1 ⟹ *denormalize* (*normalize-full t1*) ≠ []
⟨*proof*⟩

lemma *take1-eq-denormalize-full*:
wf-dlverts t1 ⟹ *take 1* (*denormalize* (*normalize-full t1*)) = [*hd* (*Dtree.root t1*)]
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *normalize-full-verts-sublist*:
v ∈ *dverts t1* ⟹ ∃ *v2* ∈ *dverts* (*normalize-full t1*). *sublist v v2*
⟨*proof*⟩

lemma *normalize-full-sublist-preserv*:
⟦*sublist xs v*; *v* ∈ *dverts t1*⟧ ⟹ ∃ *v2* ∈ *dverts* (*normalize-full t1*). *sublist xs v2*
⟨*proof*⟩

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*⟩

corollary *denormalize-sublist-preserv*:
⟦*sublist xs v*; *v* ∈ *dverts* (*t1*::('*a list*,'*b*) *dtree*); *max-deg t1* ≤ *1*⟧
⟹ *sublist xs* (*denormalize t1*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *P-denormalize-ikkbz-sub*:
assumes *hd* (*Dtree.root t*) = *root*
shows *unique-set-r root* (*dverts t*) (*denormalize* (*ikkbz-sub t*))
⟨*proof*⟩

lemma *merge1-sublist-preserv*:
⟦*sublist xs v*; *v* ∈ *dverts t*⟧ ⟹ ∃ *v2* ∈ *dverts* (*merge1 t*). *sublist xs v2*
⟨*proof*⟩

lemma *normalize1-verts-sublist*: *v* ∈ *dverts t1* ⟹ ∃ *v2* ∈ *dverts* (*normalize1 t1*). *sublist v v2*
⟨*proof*⟩

lemma *normalize1-sublist-preserv*:
⟦*sublist xs v*; *v* ∈ *dverts t1*⟧ ⟹ ∃ *v2* ∈ *dverts* (*normalize1 t1*). *sublist xs v2*
⟨*proof*⟩

lemma *normalize-verts-sublist*: *v* ∈ *dverts t1* ⟹ ∃ *v2* ∈ *dverts* (*normalize t1*). *sublist v v2*
⟨*proof*⟩

lemma *normalize-sublist-preserv*:
⟦*sublist xs v*; *v* ∈ *dverts t1*⟧ ⟹ ∃ *v2* ∈ *dverts* (*normalize t1*). *sublist xs v2*
⟨*proof*⟩

lemma *ikkbz-sub-verts-sublist*: *v* ∈ *dverts t* ⟹ ∃ *v2* ∈ *dverts* (*ikkbz-sub t*). *sublist v v2*
⟨*proof*⟩

lemma *ikkbz-sub-sublist-preserv*:
⟦*sublist xs v*; *v* ∈ *dverts t*⟧ ⟹ ∃ *v2* ∈ *dverts* (*ikkbz-sub t*). *sublist xs v2*
⟨*proof*⟩

lemma *denormalize-ikkbz-sub-verts-sublist*:
∀ *xs* ∈ (*dverts t*). *sublist xs* (*denormalize* (*ikkbz-sub t*))
⟨*proof*⟩

lemma *denormalize-ikkbz-sub-sublist-preserv*:
⟦*sublist xs v*; *v* ∈ *dverts t*⟧ ⟹ *sublist xs* (*denormalize* (*ikkbz-sub t*))
⟨*proof*⟩

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*))
⟨*proof*⟩

10.4.2 Minimal Cost of the result

lemma *normalize1-dverts-app-before-contr*:

$[\![v \in dverts\ (normalize1\ t);\ v \notin dverts\ t]\!]$
$\Longrightarrow \exists\, v1 \in dverts\ t.\ \exists\, v2 \in dverts\ t.\ v1\ @\ v2 = v \wedge before\ v1\ v2 \wedge rank\ (rev\ v2) < rank\ (rev\ v1)$
⟨*proof*⟩

lemma *normalize1-dverts-app-bfr-cntr-rnks*:
assumes $v \in dverts\ (normalize1\ t)$ **and** $v \notin dverts\ t$
shows $\exists\, U \in dverts\ t.\ \exists\, V \in dverts\ t.\ U\ @\ V = v \wedge before\ U\ V \wedge rank\ (rev\ V) < rank\ (rev\ U)$
$\wedge\ (\forall\, xs \in dverts\ t.\ (\exists\, y \in set\ xs.\ \neg\ (\exists\, x' \in set\ V.\ x' \rightarrow^+_T y) \wedge (\exists\, x \in set\ U.\ x \rightarrow^+_T y) \wedge xs \neq U) \longrightarrow rank\ (rev\ V) \leq rank\ (rev\ xs))$
⟨*proof*⟩

lemma *normalize1-dverts-app-bfr-cntr-rnks'*:
assumes $v \in dverts\ (normalize1\ t)$ **and** $v \notin dverts\ t$
shows $\exists\, U \in dverts\ t.\ \exists\, V \in dverts\ t.\ U\ @\ V = v \wedge before\ U\ V \wedge rank\ (rev\ V) \leq rank\ (rev\ U)$
$\wedge\ (\forall\, xs \in dverts\ t.\ (\exists\, y \in set\ xs.\ \neg\ (\exists\, x' \in set\ V.\ x' \rightarrow^+_T y) \wedge (\exists\, x \in set\ U.\ x \rightarrow^+_T y) \wedge xs \neq U) \longrightarrow rank\ (rev\ V) \leq rank\ (rev\ xs))$
⟨*proof*⟩

lemma *normalize1-dverts-split*:
$dverts\ (normalize1\ t1)$
$= \{v \in dverts\ (normalize1\ t1).\ v \notin dverts\ t1\} \cup \{v \in dverts\ (normalize1\ t1).\ v \in dverts\ t1\}$
⟨*proof*⟩

lemma *normalize1-dlverts-split*:
$dlverts\ (normalize1\ t1)$
$= \bigcup (set\ `\ \{v \in dverts\ (normalize1\ t1).\ v \notin dverts\ t1\})$
$\cup \bigcup (set\ `\ \{v \in dverts\ (normalize1\ t1).\ v \in dverts\ t1\})$
⟨*proof*⟩

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*⟩

lemma *normalize1-dsjnt-subset-split1*:
fixes *t1*
defines $X \equiv \{v \in dverts\ (normalize1\ t1).\ v \notin dverts\ t1\}$
assumes *wf-dlverts t1*
shows $\{x.\ x \in dverts\ t1 \wedge set\ x \cap \bigcup (set\ `\ X) = \{\}\} \subseteq \{v \in dverts\ (normalize1\ t1).\ v \in dverts\ t1\}$
⟨*proof*⟩

lemma *normalize1-dsjnt-subset-split2*:
fixes *t1*
defines $X \equiv \{v \in dverts\ (normalize1\ t1).\ v \notin dverts\ t1\}$
assumes *wf-dlverts t1*
shows $\{v \in dverts\ (normalize1\ t1).\ v \in dverts\ t1\} \subseteq \{x.\ x \in dverts\ t1 \wedge set\ x \cap \bigcup (set\ `\ X) = \{\}\}$
⟨*proof*⟩

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 $\{v \in dverts\ (normalize1\ t1).\ v \in dverts\ t1\} = \{x.\ x \in dverts\ t1 \wedge set\ x \cap \bigcup (set\ `\ X) = \{\}\}$
⟨*proof*⟩

lemma *normalize1-dverts-split2*:
fixes *t1*
defines $X \equiv \{v \in dverts\ (normalize1\ t1).\ v \notin dverts\ t1\}$
assumes *wf-dlverts t1*
shows $X \cup \{x.\ x \in dverts\ t1 \wedge set\ x \cap \bigcup (set\ `\ X) = \{\}\} = dverts\ (normalize1\ t1)$
⟨*proof*⟩

lemma *set-subset-if-normalize1-vert*: $v1 \in dverts\ (normalize1\ t1) \Longrightarrow set\ v1 \subseteq dlverts\ t1$
⟨*proof*⟩

lemma *normalize1-new-verts-not-reach1*:
assumes $v1 \in dverts\ (normalize1\ t)$ **and** $v1 \notin dverts\ t$
and $v2 \in dverts\ (normalize1\ t)$ **and** $v2 \notin dverts\ t$
and $v1 \neq v2$
shows $\neg(\exists\, x{\in}set\ v1.\ \exists\, y{\in}set\ v2.\ x \rightarrow^{+}{}_{T}\ y)$
⟨*proof*⟩

lemma *normalize1-dverts-split-optimal*:
defines $X \equiv \{v \in dverts\ (normalize1\ t).\ v \notin dverts\ t\}$
assumes $\exists\, x.\ fwd\text{-}sub\ root\ (dverts\ t)\ x$
shows $\exists\, zs.\ fwd\text{-}sub\ root\ (X \cup \{x.\ x \in dverts\ t \wedge set\ x \cap \bigcup (set\ `\ X) = \{\}\})\ zs$
$\wedge\ (\forall\, as.\ fwd\text{-}sub\ root\ (dverts\ t)\ as \longrightarrow cost\ (rev\ zs) \leq cost\ (rev\ as))$
⟨*proof*⟩

corollary *normalize1-dverts-optimal*:
assumes $\exists\, x.\ fwd\text{-}sub\ root\ (dverts\ t)\ x$
shows $\exists\, zs.\ fwd\text{-}sub\ root\ (dverts\ (normalize1\ t))\ zs$
$\wedge\ (\forall\, as.\ fwd\text{-}sub\ root\ (dverts\ t)\ as \longrightarrow cost\ (rev\ zs) \leq cost\ (rev\ as))$
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *ikkbz-sub-dverts-optimal'*:
assumes *hd* (*Dtree.root t*) = *root* **and** *max-deg t* $\leq$ *1* $\Longrightarrow$ *dom-children t T*
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*))
⟨*proof*⟩

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*⟩

lemma *combine-subtree-orig-uneq*:
assumes *is-subtree* (*Node r1* {|(*t2*,*e2*)|}) (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
shows *Node r1* {|(*t2*,*e2*)|} $\neq$ *Node r* {|(*t1*,*e1*)|}
⟨*proof*⟩

lemma *combine-strict-subtree-ranks-le*:
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 *strict-subtree* (*Node r1* {|(*t2*,*e2*)|}) (*Node* (*r*@*Dtree.root t1*) (*sucs t1*))
shows *rank* (*rev r1*) $\leq$ *rank* (*rev* (*Dtree.root t2*))
⟨*proof*⟩

lemma *subtree-child-uneq*:
⟦*is-subtree t1 t2*; *t2* $\in$ *fst* ‘ *fset xs*⟧ $\Longrightarrow$ *t1* $\neq$ *Node r xs*
⟨*proof*⟩

lemma *subtree-singleton-child-uneq*:
is-subtree t1 t2 $\Longrightarrow$ *t1* $\neq$ *Node r* {|(*t2*,*e2*)|}
⟨*proof*⟩

lemma *child-subtree-ranks-le-if-strict-subtree*:
assumes $\bigwedge r1\ t2\ e2.\ \mathit{strict\text{-}subtree}\ (\mathit{Node}\ r1\ \{|(t2,e2)|\})\ (\mathit{Node}\ r\ \{|(t1,e1)|\}) \Longrightarrow \mathit{rank}\ (\mathit{rev}\ r1) \leq \mathit{rank}\ (\mathit{rev}\ (\mathit{Dtree.root}\ t2))$
and $\mathit{is\text{-}subtree}\ (\mathit{Node}\ r1\ \{|(t2,e2)|\})\ t1$
shows $\mathit{rank}\ (\mathit{rev}\ r1) \leq \mathit{rank}\ (\mathit{rev}\ (\mathit{Dtree.root}\ t2))$
⟨*proof*⟩

lemma *verts-ge-child-if-sorted*:
assumes $\bigwedge r1\ t2\ e2.\ \mathit{strict\text{-}subtree}\ (\mathit{Node}\ r1\ \{|(t2,e2)|\})\ (\mathit{Node}\ r\ \{|(t1,e1)|\}) \Longrightarrow \mathit{rank}\ (\mathit{rev}\ r1) \leq \mathit{rank}\ (\mathit{rev}\ (\mathit{Dtree.root}\ t2))$
and $\mathit{max\text{-}deg}\ (\mathit{Node}\ r\ \{|(t1,e1)|\}) \leq 1$
and $v \in \mathit{dverts}\ t1$
shows $\mathit{rank}\ (\mathit{rev}\ (\mathit{Dtree.root}\ t1)) \leq \mathit{rank}\ (\mathit{rev}\ v)$
⟨*proof*⟩

lemma *verts-ge-child-if-sorted'*:
assumes $\bigwedge r1\ t2\ e2.\ \mathit{strict\text{-}subtree}\ (\mathit{Node}\ r1\ \{|(t2,e2)|\})\ (\mathit{Node}\ r\ \{|(t1,e1)|\}) \Longrightarrow \mathit{rank}\ (\mathit{rev}\ r1) \leq \mathit{rank}\ (\mathit{rev}\ (\mathit{Dtree.root}\ t2))$
and $\mathit{max\text{-}deg}\ (\mathit{Node}\ r\ \{|(t1,e1)|\}) \leq 1$
and $v \in \mathit{dverts}\ (\mathit{Node}\ r\ \{|(t1,e1)|\})$
and $v \neq r$
shows $\mathit{rank}\ (\mathit{rev}\ (\mathit{Dtree.root}\ t1)) \leq \mathit{rank}\ (\mathit{rev}\ v)$
⟨*proof*⟩

lemma *not-combined-sub-dverts-combine*:
$\{r@\mathit{Dtree.root}\ t1\} \cup \{x.\ x \in \mathit{dverts}\ (\mathit{Node}\ r\ \{|(t1,e1)|\}) \wedge x \neq r \wedge x \neq \mathit{Dtree.root}\ t1\}$
$\subseteq \mathit{dverts}\ (\mathit{Node}\ (r\ @\ \mathit{Dtree.root}\ t1)\ (\mathit{sucs}\ t1))$
⟨*proof*⟩

lemma *dverts-combine-orig-not-combined*:
assumes $\mathit{wf\text{-}dlverts}\ (\mathit{Node}\ r\ \{|(t1,e1)|\})$ **and** $x \in \mathit{dverts}\ (\mathit{Node}\ (r\ @\ \mathit{Dtree.root}\ t1)\ (\mathit{sucs}\ t1))$ **and** $x \neq r@\mathit{Dtree.root}\ t1$
shows $x \in \mathit{dverts}\ (\mathit{Node}\ r\ \{|(t1,e1)|\}) \wedge x \neq r \wedge x \neq \mathit{Dtree.root}\ t1$
⟨*proof*⟩

lemma *dverts-combine-sub-not-combined*:
$\mathit{wf\text{-}dlverts}\ (\mathit{Node}\ r\ \{|(t1,e1)|\}) \Longrightarrow \mathit{dverts}\ (\mathit{Node}\ (r\ @\ \mathit{Dtree.root}\ t1)\ (\mathit{sucs}\ t1))$
$\subseteq \{r@\mathit{Dtree.root}\ t1\} \cup \{x.\ x \in \mathit{dverts}\ (\mathit{Node}\ r\ \{|(t1,e1)|\}) \wedge x \neq r \wedge x \neq \mathit{Dtree.root}\ t1\}$
⟨*proof*⟩

lemma *dverts-combine-eq-not-combined*:
$\mathit{wf\text{-}dlverts}\ (\mathit{Node}\ r\ \{|(t1,e1)|\}) \Longrightarrow \mathit{dverts}\ (\mathit{Node}\ (r\ @\ \mathit{Dtree.root}\ t1)\ (\mathit{sucs}\ t1))$
$= \{r@\mathit{Dtree.root}\ t1\} \cup \{x.\ x \in \mathit{dverts}\ (\mathit{Node}\ r\ \{|(t1,e1)|\}) \wedge x \neq r \wedge x \neq \mathit{Dtree.root}\ t1\}$
⟨*proof*⟩

lemma *normalize-full-dverts-optimal-if-sorted*:

assumes *asi rank root cost*
and *wf-dlverts t1*
and $\forall$ *xs* $\in$ (*dverts t1*). *distinct xs*
and $\forall$ *xs* $\in$ (*dverts t1*). *seq-conform xs*
and $\bigwedge$*r1 t2 e2*. *strict-subtree* (*Node r1* {|(*t2*,*e2*)|}) *t1*
$\Longrightarrow$ *rank* (*rev r1*) $\leq$ *rank* (*rev* (*Dtree.root t2*))
and *max-deg t1* $\leq$ *1*
and *hd* (*Dtree.root t1*) = *root*
and *dom-children t1 T*
shows $\exists$ *zs*. *fwd-sub root* (*dverts* (*normalize-full t1*)) *zs*
$\wedge$ ($\forall$ *as*. *fwd-sub root* (*dverts t1*) *as* $\longrightarrow$ *cost* (*rev zs*) $\leq$ *cost* (*rev as*))
⟨*proof*⟩

corollary *normalize-full-dverts-optimal-if-sorted′*:
assumes *max-deg t* $\leq$ *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*))
⟨*proof*⟩

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*⟩

lemma *single-set-distinct-sublist*: ⟦*set ys* = *set x*; *distinct ys*; *sublist x ys*⟧ $\Longrightarrow$ *x* = *ys*
⟨*proof*⟩

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*⟩

theorem *denormalize-ikkbz-sub-optimal*:
assumes *hd* (*Dtree.root t*) = *root* **and** *max-deg t* $\leq$ *1* $\Longrightarrow$ *dom-children t T*
shows ($\forall$ *as*. *fwd-sub root* (*dverts t*) *as*
$\longrightarrow$ *cost* (*rev* (*denormalize* (*ikkbz-sub t*))) $\leq$ *cost* (*rev as*))
⟨*proof*⟩

end

10.5 Arc Invariants hold for Conversion to Dtree

context *precedence-graph*
begin

interpretation *t*: *ranked-dtree to-list-dtree* ⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *to-list-dtree-nempty-root*: *is-subtree* (*Node r xs*) *to-list-dtree* ⟹ *r* ≠ []
⟨*proof*⟩

lemma *dom-children-aux*:
assumes *is-subtree* (*Node r xs*) *to-list-dtree*
and *max-deg t1* ≤ *1*
and (*t1*,*e1*) ∈ *fset xs*
and *x* ∈ *dlverts t1*
shows ∃ *v* ∈ *set r* ∪ *path-lverts t1 x*. *v* →$_T$ *x*
⟨*proof*⟩

lemma *hd-dverts-in-dlverts*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; (*t1*,*e1*) ∈ *fset xs*; *x* ∈ *dverts t1*⟧ ⟹ *hd x* ∈ *dlverts t1*
⟨*proof*⟩

lemma *dom-children-aux2*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; *max-deg t1* ≤ *1*; (*t1*,*e1*) ∈ *fset xs*; *x* ∈ *dverts t1*⟧
⟹ ∃ *v* ∈ *set r* ∪ *path-lverts t1* (*hd x*). *v* →$_T$ (*hd x*)
⟨*proof*⟩

lemma *dom-children-full*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; ∀ *t* ∈ *fst* ‘ *fset xs*. *max-deg t* ≤ *1*⟧
⟹ *dom-children* (*Node r xs*) *T*
⟨*proof*⟩

lemma *dom-children′*:
assumes *is-subtree* (*Node r xs*) *to-list-dtree*
shows *dom-children* (*Node r* (*Abs-fset* (*children-deg1 xs*))) *T*
⟨*proof*⟩

lemma *dom-children-maxdeg-1*:

⟦*is-subtree* (*Node r xs*) *to-list-dtree*; *max-deg* (*Node r xs*) ≤ *1*⟧
⟹ *dom-children* (*Node r xs*) *T*
⟨*proof*⟩

lemma *dom-child-subtree*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; *t* ∈ *fst* ‘ *fset xs*⟧ ⟹ ∃ *v*∈*set r*. *v* →$_T$ *hd* (*Dtree.root t*)
⟨*proof*⟩

lemma *dom-children-maxdeg-1-self*:
max-deg to-list-dtree ≤ *1* ⟹ *dom-children to-list-dtree T*
⟨*proof*⟩

lemma *seq-conform-list-tree*: ∀ *v*∈*verts to-list-tree*. *seq-conform v*
⟨*proof*⟩

lemma *conform-list-dtree*: ∀ *v*∈*dverts to-list-dtree*. *seq-conform v*
⟨*proof*⟩

lemma *to-list-dtree-vert-single*: ⟦*v* ∈ *dverts to-list-dtree*; *x* ∈ *set v*⟧ ⟹ *v* = [*x*] ∧ *x* ∈ *verts T*
⟨*proof*⟩

lemma *to-list-dtree-vert-single-sub*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; *x* ∈ *set r*⟧ ⟹ *r* = [*x*] ∧ *x* ∈ *verts T*
⟨*proof*⟩

lemma *to-list-dtree-child-if-to-list-tree-arc*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; *r* →$_{to\text{-}list\text{-}tree}$ *v*⟧ ⟹ ∃ *ys*. (*Node v ys*) ∈ *fst* ‘ *fset xs*
⟨*proof*⟩

lemma *to-list-dtree-child-if-arc*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; *x* ∈ *set r*; *x* →$_T$ *y*⟧
⟹ ∃ *ys*. *Node* [*y*] *ys* ∈ *fst* ‘ *fset xs*
⟨*proof*⟩

lemma *to-list-dtree-dverts-if-arc*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; *x* ∈ *set r*; *x* →$_T$ *y*⟧ ⟹ [*y*] ∈ *dverts* (*Node r xs*)
⟨*proof*⟩

lemma *to-list-dtree-dlverts-if-arc*:
⟦*is-subtree* (*Node r xs*) *to-list-dtree*; *x* ∈ *set r*; *x* →$_T$ *y*⟧ ⟹ *y* ∈ *dlverts* (*Node r xs*)
⟨*proof*⟩

theorem *to-list-dtree-ranked-orig*: *ranked-dtree-with-orig to-list-dtree rank cost cmp T root*

⟨*proof*⟩

interpretation *t*: *ranked-dtree-with-orig to-list-dtree* ⟨*proof*⟩

lemma *forward-ikkbz-sub*: *forward ikkbz-sub*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

lemma *hd-eq-tk1-if-set-eq-verts*: *set xs* = *verts T* ⟹ *hd xs* = *root* ⟷ *take 1 xs* = [*root*]
⟨*proof*⟩

lemma *ikkbz-sub-optimal*:
⟦*set xs* = *verts T*; *distinct xs*; *forward xs*; *hd xs* = *root*⟧
⟹ *cost* (*rev ikkbz-sub*) ≤ *cost* (*rev xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ikkbz-sub-no-cross*: *r* ∈ *verts G* ⟹ *no-cross-products* (*create-ldeep* (*ikkbz-sub r*))
⟨*proof*⟩

lemma *ikkbz-sub-cost-r-eq-cost*:
r ∈ *verts G* ⟹ *cost-r r* (*rev* (*ikkbz-sub r*)) = *cost-l* (*ikkbz-sub r*)

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ikkbz-no-cross*: *no-cross-products* (*create-ldeep ikkbz*)
⟨*proof*⟩

lemma *hd-in-verts-if-set-eq*: *set xs* = *verts G* ⟹ *hd xs* ∈ *verts G*
⟨*proof*⟩

lemma *ikkbz-optimal*:
⟦*set xs* = *verts G*; *distinct xs*; *no-cross-products* (*create-ldeep xs*)⟧
⟹ *cost-l ikkbz* ≤ *cost-l xs*
⟨*proof*⟩

theorem *ikkbz-optimal-tree*:
⟦*valid-tree t*; *no-cross-products t*; *left-deep t*⟧ ⟹ *cost* (*create-ldeep ikkbz*) ≤ *cost t*
⟨*proof*⟩

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* ⟹ *contr-sel sf x* > *0*
⟨*proof*⟩

lemma *contr-sel-le1*: *sel-reasonable sf* $\Longrightarrow$ *contr-sel sf x* $\leq$ *1*
⟨*proof*⟩

lemma *nempty-if-not-fwd-conc*: ¬*forward-arcs* (*y*#*xs*) $\Longrightarrow$ *xs* $\neq$ []
⟨*proof*⟩

lemma *len-gt1-if-not-fwd-conc*: ¬*forward-arcs* (*y*#*xs*) $\Longrightarrow$ *length* (*y*#*xs*) > *1*
⟨*proof*⟩

lemma *two-elems-if-not-fwd-conc*: ¬*forward-arcs* (*y*#*xs*) $\Longrightarrow$ $\exists$ *a b cs. a* # *b* # *cs* = *y*#*xs*
⟨*proof*⟩

lemma *hd-reach-all-if-nfwd-app-fwd*:
⟦¬*forward-arcs* (*y*#*xs*); *forward-arcs* (*y*#*ys*@*xs*); *x* $\in$ *set* (*y*#*ys*@*xs*)⟧
$\Longrightarrow$ *hd* (*rev* (*y*#*ys*@*xs*)) $\rightarrow^*{}_T$ *x*
⟨*proof*⟩

lemma *hd-not-y-if-if-nfwd-app-fwd*:
assumes ¬*forward-arcs* (*y*#*xs*) **and** *forward-arcs* (*y*#*ys*@*xs*)
shows *hd* (*rev* (*y*#*ys*@*xs*)) $\neq$ *y*
⟨*proof*⟩

lemma *hd-reach1-y-if-nfwd-app-fwd*:
⟦¬*forward-arcs* (*y*#*xs*); *forward-arcs* (*y*#*ys*@*xs*)⟧ $\Longrightarrow$ *hd* (*rev* (*y*#*ys*@*xs*)) $\rightarrow^+{}_T$ *y*
⟨*proof*⟩

lemma *not-fwd-if-skip1*:
⟦¬ *forward-arcs* (*y*#*x*#*x'*#*xs*); *forward-arcs* (*x*#*x'*#*xs*)⟧ $\Longrightarrow$ ¬ *forward-arcs* (*y*#*x'*#*xs*)
⟨*proof*⟩

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*⟩

lemma *fwd-app-nhead-elem*: ⟦*forward xs*; *y* $\in$ *set xs*; *y* $\neq$ *hd xs*⟧ $\Longrightarrow$ *forward* (*xs*@[*y*])
⟨*proof*⟩

lemma *hd-last-not-fwd-arcs*: ¬*forward-arcs* (*x*#*xs*@[*x*])
⟨*proof*⟩

lemma *hd-not-fwd-arcs*: ¬*forward-arcs* (*ys*@*x*#*xs*@[*x*])
⟨*proof*⟩

lemma *hd-last-not-fwd*: ¬*forward* (*x*#*xs*@[*x*])

⟨*proof*⟩

lemma *hd-not-fwd*: ¬*forward* (*x*#*xs*@[*x*]@*ys*)
⟨*proof*⟩

lemma *y-not-dom-if-nfwd-app-fwd*:
⟦¬*forward-arcs* (*y*#*xs*); *forward-arcs* (*y*#*ys*@*xs*); *x* ∈ *set xs*⟧ ⟹ ¬ *x* →$_T$ *y*
⟨*proof*⟩

lemma *not-y-dom-if-nfwd-app-fwd*:
⟦¬*forward-arcs* (*y*#*xs*); *forward-arcs* (*y*#*ys*@*xs*); *x* ∈ *set xs*⟧ ⟹ ¬ *y* →$_T$ *x*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *contr-sel-root*[*simp*]: *contr-sel sel root* = *1*
⟨*proof*⟩

lemma *contr-sel-notvert*[*simp*]: *v* ∉ *verts T* ⟹ *contr-sel sel v* = *1*
⟨*proof*⟩

lemma *hd-reach-all-forward-verts*:
⟦*forward xs*; *set xs* = *verts T*; *v* ∈ *verts T*⟧ ⟹ *hd xs* →*$_T$ *v*
⟨*proof*⟩

lemma *hd-eq-root-if-forward-verts*: ⟦*forward xs*; *set xs* = *verts T*⟧ ⟹ *hd xs* = *root*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *dst-fwd-all-verts-ex*: ∃ *xs*. *forward xs* ∧ *distinct xs* ∧ *set xs* = *verts T*
⟨*proof*⟩

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*)
⟨*proof*⟩

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**

$cf'\ x = (\mathit{if}\ x \in \mathit{verts}\ G\ \mathit{then}\ cf\ x\ \mathit{else}\ 1)$

definition *c-list-r* :: $('a \Rightarrow \mathit{real}) \Rightarrow 'a \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{real}$ **where**
$\mathit{c\text{-}list\text{-}r}\ h\ r = \mathit{c\text{-}list}\ (\mathit{sel\text{-}r}\ r)\ cf'\ h\ r$

definition *rank-r* :: $('a \Rightarrow \mathit{real}) \Rightarrow 'a \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{real}$ **where**
$\mathit{rank\text{-}r}\ h\ r\ xs = (\mathit{ldeep\text{-}T}\ (\mathit{sel\text{-}r}\ r)\ cf'\ xs - 1)\ /\ \mathit{c\text{-}list\text{-}r}\ h\ r\ xs$

lemma *dom-in-dir-tree-r*:
assumes $r \in \mathit{verts}\ G$ **and** $x \rightarrow_G y$
shows $x \rightarrow_{\mathit{dir\text{-}tree\text{-}r}\ r} y \vee y \rightarrow_{\mathit{dir\text{-}tree\text{-}r}\ r} x$
⟨*proof*⟩

lemma *dom-in-dir-tree-r-iff-aux*:
$r \in \mathit{verts}\ G \Longrightarrow (x \rightarrow_{\mathit{dir\text{-}tree\text{-}r}\ r} y \vee y \rightarrow_{\mathit{dir\text{-}tree\text{-}r}\ r} x) \longleftrightarrow (x \rightarrow_G y \vee y \rightarrow_G x)$
⟨*proof*⟩

lemma *dom-in-dir-tree-r-iff*:
$r \in \mathit{verts}\ G \Longrightarrow (x \rightarrow_{\mathit{dir\text{-}tree\text{-}r}\ r} y \vee y \rightarrow_{\mathit{dir\text{-}tree\text{-}r}\ r} x) \longleftrightarrow x \rightarrow_G y$
⟨*proof*⟩

lemma *dir-tree-sel*[*intro*]: $r \in \mathit{verts}\ G \Longrightarrow \mathit{directed\text{-}tree.tree\text{-}sel}\ (\mathit{dir\text{-}tree\text{-}r}\ r)\ \mathit{match\text{-}sel}$
⟨*proof*⟩

lemma *pos-cards'*[*intro*!]: $\forall x.\ cf'\ x > 0$
⟨*proof*⟩

theorem *c-list-asi*: $[\![r \in \mathit{verts}\ G; \forall x.\ h\ x > 0]\!] \Longrightarrow \mathit{asi}\ (\mathit{rank\text{-}r}\ h\ r)\ r\ (\mathit{c\text{-}list\text{-}r}\ h\ r)$
⟨*proof*⟩

11.3 Applying IKKBZ

lemma *cf'-simp*: $x \in \mathit{verts}\ G \Longrightarrow cf'\ x = cf\ x$
⟨*proof*⟩

lemma *ldeep-T-cf'-eq*: $\mathit{set}\ xs \subseteq \mathit{verts}\ G \Longrightarrow \mathit{ldeep\text{-}T}\ sf\ cf'\ xs = \mathit{ldeep\text{-}T}\ sf\ cf\ xs$
⟨*proof*⟩

lemma *clist-cf'-eq*: $\mathit{set}\ xs \subseteq \mathit{verts}\ G \Longrightarrow \mathit{c\text{-}list}\ sf\ cf'\ h\ r\ xs = \mathit{c\text{-}list}\ sf\ cf\ h\ r\ xs$
⟨*proof*⟩

lemma *card-cf'-eq*: $\mathit{matching\text{-}rels}\ t \Longrightarrow \mathit{card}\ cf'\ f\ t = \mathit{card}\ cf\ f\ t$
⟨*proof*⟩

lemma *c-IKKBZ-cf'-eq*: $\mathit{matching\text{-}rels}\ t \Longrightarrow \mathit{c\text{-}IKKBZ}\ h\ cf'\ sf\ t = \mathit{c\text{-}IKKBZ}\ h\ cf\ sf\ t$
⟨*proof*⟩

lemma *c-IKKBZ-cf'-eq'*: *valid-tree t* $\Longrightarrow$ *c-IKKBZ h cf' sf t* = *c-IKKBZ h cf sf t*
⟨*proof*⟩

lemma *c-out-cf'-eq*: *matching-rels t* $\Longrightarrow$ *c-out cf' sf t* = *c-out cf sf t*
⟨*proof*⟩

lemma *c-out-cf'-eq'*: *valid-tree t* $\Longrightarrow$ *c-out cf' sf t* = *c-out cf sf t*
⟨*proof*⟩

lemma *joinTree-card'-pos*[*intro*]: *pos-rel-cards cf' t*
⟨*proof*⟩

lemma *match-reasonable-cards'*[*intro*]: *reasonable-cards cf' match-sel t*
⟨*proof*⟩

lemma *sel-r-gt0*: *r* ∈ *verts G* $\Longrightarrow$ *sel-r r x* > *0*
⟨*proof*⟩

lemma *sel-r-le1*: *r* ∈ *verts G* $\Longrightarrow$ *sel-r r x* ≤ *1*
⟨*proof*⟩

lemma *sel-r-eq-ldeep-s-if-dst-fwd-verts*:
⟦*r* ∈ *verts G*; *distinct xs*; *directed-tree.forward* (*dir-tree-r r*) *xs*; *set xs* = *verts G*⟧
$\Longrightarrow$ *sel-r r* = *ldeep-s match-sel* (*rev xs*)
⟨*proof*⟩

lemma *sel-r-eq-ldeep-s-if-valid-fwd*:
⟦*r* ∈ *verts G*; *valid-tree t*; *directed-tree.forward* (*dir-tree-r r*) (*inorder t*)⟧
$\Longrightarrow$ *sel-r r* = *ldeep-s match-sel* (*revorder t*)
⟨*proof*⟩

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* (*revorder t*)
⟨*proof*⟩

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* (*revorder t*)) *cf' h* (*first-node t*) *xs*
= *c-list-r h* (*first-node t*) *xs*
⟨*proof*⟩

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* (λ*x*. *h x* (*cf' x*)) (*first-node t*) (*revorder t*) = *c-IKKBZ h cf match-sel t*
⟨*proof*⟩

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 $h :: 'a \Rightarrow real \Rightarrow real$
  assumes *h-pos*: $\forall x.\ h\ x\ (cf'\ x) > 0$
begin

theorem *ikkbz-query-graph-if-simple-h*:
  defines $cost \equiv$ *c-IKKBZ h cf match-sel*
  defines $h' \equiv (\lambda x.\ h\ x\ (cf'\ x))$
  shows *ikkbz-query-graph bfs sel cf G cmp cost* (*c-list-r h'*) (*rank-r h'*)
  ⟨*proof*⟩

interpretation *ikkbz-query-graph bfs sel cf G cmp*
    c-IKKBZ h cf match-sel c-list-r $(\lambda x.\ h\ x\ (cf'\ x))$ *rank-r* $(\lambda x.\ h\ x\ (cf'\ x))$
  ⟨*proof*⟩

corollary *ikkbz-simple-h-nempty*: *ikkbz* $\neq$ []
  ⟨*proof*⟩

corollary *ikkbz-simple-h-valid-tree*: *valid-tree* (*create-ldeep ikkbz*)
  ⟨*proof*⟩

corollary *ikkbz-simple-h-no-cross*:
  no-cross-products (*create-ldeep ikkbz*)
  ⟨*proof*⟩

theorem *ikkbz-simple-h-optimal*:
  ⟦*valid-tree t*; *no-cross-products t*; *left-deep t*⟧
    $\Longrightarrow$ *c-IKKBZ h cf match-sel* (*create-ldeep ikkbz*) $\leq$ *c-IKKBZ h cf match-sel t*
  ⟨*proof*⟩

abbreviation *ikkbz-simple-h* :: $'a$ *list* **where**
  ikkbz-simple-h $\equiv$ *ikkbz*
end

We can now apply these results directly to valid cost functions like *c-nlj* and *c-hj*.

lemma *id-cf'-gt0*: $\forall x.\ id\ (cf'\ x) > 0$
  ⟨*proof*⟩

corollary *ikkbz-nempty-nlj*: *ikkbz-simple-h* $(\lambda\text{-}.\ id) \neq []$
  ⟨*proof*⟩

corollary *ikkbz-valid-tree-nlj*: *valid-tree* (*create-ldeep* (*ikkbz-simple-h* (λ-. *id*)))
⟨*proof*⟩

corollary *ikkbz-no-cross-nlj*: *no-cross-products* (*create-ldeep* (*ikkbz-simple-h* (λ-. *id*)))
⟨*proof*⟩

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*
⟨*proof*⟩

corollary *ikkbz-nempty-hj*: *ikkbz-simple-h* (λ- -. *1.2*) ≠ []
⟨*proof*⟩

corollary *ikkbz-valid-tree-hj*: *valid-tree* (*create-ldeep* (*ikkbz-simple-h* (λ- -. *1.2*)))
⟨*proof*⟩

corollary *ikkbz-no-cross-hj*: *no-cross-products* (*create-ldeep* (*ikkbz-simple-h* (λ- -. *1.2*)))
⟨*proof*⟩

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*
⟨*proof*⟩

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 $h \equiv (\lambda a.\ \mathit{ldeep\text{-}s}\ \mathit{match\text{-}sel}\ \mathit{xs}\ a * \mathit{cf}'\ a)$
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*
⟨*proof*⟩

lemma *c-out-list-correct-cf′*:
fixes *t*
defines $h \equiv (\lambda a.\ \mathit{sel\text{-}r}\ (\mathit{first\text{-}node}\ t)\ a * \mathit{cf}'\ a)$
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*
⟨*proof*⟩

lemma *c-out-list-correct-cf*:
fixes *t*
defines $h \equiv (\lambda a.\ \mathit{sel\text{-}r}\ (\mathit{first\text{-}node}\ t)\ a * \mathit{cf}'\ a)$
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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *c-out-h-gt0*: *r* ∈ *verts G* ⟹ $(\lambda a.\ \mathit{sel\text{-}r}\ r\ a * \mathit{cf}'\ a)\ x > 0$
⟨*proof*⟩

lemma *c-out-r-asi*: *r* ∈ *verts G* ⟹ *asi* (*c-out-rank-r r*) *r* (*c-out-list-r r*)
⟨*proof*⟩

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*
⟨*proof*⟩

interpretation QG_{out}:
ikkbz-query-graph bfs sel cf G cmp c-out cf match-sel c-out-list-r c-out-rank-r
⟨*proof*⟩

corollary *ikkbz-nempty-cout*: $QG_{out}.\mathit{ikkbz} \neq []$
⟨*proof*⟩

corollary *ikkbz-valid-tree-cout*: *valid-tree* (*create-ldeep* $QG_{out}.\mathit{ikkbz}$)
⟨*proof*⟩

corollary *ikkbz-no-cross-cout*: *no-cross-products* (*create-ldeep* QG_{out}.*ikkbz*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *cmp-hd-eq-if-equiv*: *compare cmp* (*v1*,*e1*) (*v2*,*e2*) = *Equiv* ⟹ *hd v1* = *hd v2*
⟨*proof*⟩

lemma *cmp-sets-not-dsjnt-if-equiv*:
⟦*v1* ≠ []; *v2* ≠ []; *compare cmp* (*v1*,*e1*) (*v2*,*e2*) = *Equiv*⟧ ⟹ *set v1* ∩ *set v2* ≠ {}
⟨*proof*⟩

lemma *cmp-tree-qg*: *cmp-tree-query-graph bfs sel cf G cmp*
⟨*proof*⟩

interpretation *cmp-tree-query-graph bfs sel cf G cmp*
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *cmp-arcs-eq-if-equiv*: *compare cmp* (*v1*,*e1*) (*v2*,*e2*) = *Equiv* ⟹ *e1* = *e2*
⟨*proof*⟩

lemma *cmp-tree-qg*: *cmp-tree-query-graph bfs sel cf G cmp*
⟨*proof*⟩

interpretation *cmp-tree-query-graph bfs sel cf G cmp*
⟨*proof*⟩

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.