Proving a data flow analysis algorithm for computing dominators

Nan Jiang

March 17, 2025

Abstract

This entry formalises a fast iterative algorithm for computing dominators [1]. It gives a specification of computing dominators on a control flow graph where each node refers to its reverse post order number. A semilattice of reversed-ordered list which represents dominators is built and a Kildall's algorithm on the semilattice is defined for computing dominators. Finally the soundness and completeness of the algorithm are proved w.r.t. the specification.

Contents

1 The specification of computing dominators **1**

2 More auxiliary lemmas for Lists Sorted wrt < **20**

3 Operations on sorted lists **22**

4 A semilattice of reversed-ordered list **24**

5 A kildall's algorithm for computing dominators **29**

6 Properties of the kildall's algorithm on the semilattice **31**

7 Soundness and completeness **37**

1 The specification of computing dominators

theory *Cfg*
imports *Main*
begin

The specification of computing dominators is defined. For fast data flow analysis presented by CHK [1], a directed graph with explicit node list and sets of initial nodes is defined. Each node refers to its rPO (reverse PostOrder) number w.r.t a DFST, and related properties as assumptions are handled using a locale.

type-synonym *'a digraph* = (*'a* × *'a*) *set*

record *'a graph-rec* =
  g-V :: *'a list*
  g-V0 :: *'a set*
  g-E :: *'a digraph*

  tail :: *'a* × *'a* ⇒ *'a*
  head :: *'a* × *'a* ⇒ *'a*

definition *wf-cfg* :: *'a graph-rec* ⇒ *bool* **where**
  wf-cfg G ≡ *g-V0 G* ⊆ *set*(*g-V G*)

type-synonym *node* = *nat*

locale *cfg-doms* =
  — Nodes are rPO numbers
  fixes *G* :: *nat graph-rec* (**structure**)

  — General properties
  assumes *wf-cfg*: *wf-cfg G*
  assumes *tail*[*simp*]: *e* = (*u*,*v*) ⟹ *tail G e* = *u*
  assumes *head*[*simp*]: *e* = (*u*,*v*) ⟹ *head G e* = *v*
  assumes *tail-in-verts*[*simp*]: *e* ∈ *g-E G* ⟹ *tail G e* ∈ *set* (*g-V G*)
  assumes *head-in-verts*[*simp*]: *e* ∈ *g-E G* ⟹ *head G e* ∈ *set* (*g-V G*)

  — Properties of a cfg where nodes are rPO numbers
  assumes *entry0*: *g-V0 G* = {*0*}
  assumes *dfst*: ∀ *v* ∈ *set* (*g-V G*) − {*0*}. ∃ *prev*. (*prev*, *v*) ∈ *g-E G* ∧ *prev* < *v*
  assumes *reachable*: ∀ *v* ∈ *set* (*g-V G*). *v* ∈ (*g-E G*)* `` {*0*}
  assumes *verts*: *g-V G* = [*0* ..< (*length* (*g-V G*))]

— It is required that the entry node has an immediate successor which is not itself; Otherwise, no need to compute dominators It is required for proving the lemma: "wf_dom start (unstables r step start)".
  assumes *succ-of-entry0*: ∃ *s*. (*0*,*s*) ∈ *g-E G* ∧ *s* ≠ *0*

begin

inductive *path-entry* :: *nat digraph* ⇒ *nat list* ⇒ *nat* ⇒ *bool* **for** *E* **where**
  path-entry0: *path-entry E* [] *0*
| *path-entry-prepend*: ⟦ (*u*,*v*)∈ *E*; *path-entry E l u* ⟧ ⟹ *path-entry E* (*u*#*l*) *v*

lemma *path-entry0-empty-conv*: *path-entry E* [] *v* ⟷ *v* = *0*
 by (*auto intro*: *path-entry0 elim*: *path-entry.cases*)

inductive-cases *path-entry-uncons*: *path-entry E* (*u*′#*l*) *w*
inductive-simps *path-entry-cons-conv*: *path-entry E* (*u*′#*l*) *w*

lemma *single-path-entry*: *path-entry E* [*p*] *w* ⟹ *p* = *0*
  by (*simp add*: *path-entry-cons-conv path-entry0-empty-conv*)

lemma *path-entry-append*:
  ⟦ *path-entry E l v*; (*v*,*w*)∈*E* ⟧ ⟹ *path-entry E* (*v*#*l*) *w*
  by (*rule path-entry-prepend*)

lemma *entry-rtrancl-is-path*:
  assumes (*0*,*v*) ∈ E^*
  obtains *p* **where** *path-entry E p v*
    using *assms*
    by *induct* (*auto intro*:*path-entry0 path-entry-prepend*)

lemma *path-entry-is-trancl*:
  assumes *path-entry E l v*
  and *l* ≠ []
  shows (*0*,*v*)∈E^+
  using *assms*
  apply *induct*
  apply *auto* []
  apply (*case-tac l*)
  apply (*auto simp add*:*path-entry0-empty-conv*)
  done

lemma *tail-is-vert*: (*u*,*v*) ∈ *g-E G* ⟹ *u* ∈ *set* (*g-V G*)
  by (*auto dest*: *tail-in-verts*[*of* (*u*,*v*)])

lemma *head-is-vert*: (*u*,*v*) ∈ *g-E G* ⟹ *v* ∈ *set* (*g-V G*)
  by (*auto dest*: *head-in-verts*[*of* (*u*,*v*)])

lemma *tail-is-vert2*: (*u*,*v*) ∈ (*g-E G*)$^+$ ⟹ *u* ∈ *set* (*g-V G*)
  by (*induct rule*:*trancl.induct*)(*auto dest*: *tail-in-verts*)

lemma *head-is-vert2*: (*u*,*v*) ∈ (*g-E G*)$^+$ ⟹ *v* ∈ *set* (*g-V G*)
  by (*induct rule*:*trancl.induct*)(*auto dest*: *head-in-verts*)

lemma *verts-set*: *set* (*g-V G*) = {*0* ..< *length* (*g-V G*)}
proof−
  from *verts* **have** *set* (*g-V G*) = *set* [*0* ..< (*length* (*g-V G*))] **by** *simp*
  also have *set* [*0* ..< (*length* (*g-V G*))] = {*0* ..< (*length* (*g-V G*))} **by** *simp*
  ultimately show *?thesis* **by** *auto*
qed

lemma *fin-verts*: *finite* (*set* (*g-V G*))
  by (*auto*)

lemma *zero-in-verts*: *0* ∈ *set* (*g-V G*)
using *wf-cfg entry0* **by** (*unfold wf-cfg-def*) *auto*

lemma *verts-not-empty*: *g-V G* ≠ []
using *zero-in-verts* **by** *auto*

lemma *len-verts-gt0*: *length* (*g-V G*) > *0*
by (*simp add*:*verts-not-empty*)

lemma *len-verts-gt1*: *length* (*g-V G*) > *1*
proof−
from *succ-of-entry0* **obtain** *s* **where** *s* ∈ *set* (*g-V G*) **and** *s* ≠ *0* **using** *head-is-vert* **by** *auto*
with *zero-in-verts* **have** {*0*,*s*} ⊆ *set* (*g-V G*) **and** *c*: *card* {*0*, *s*} > *1* **by** *auto*
then have *card* {*0*, *s*} ≤ *card* (*set* (*g-V G*)) **by** (*auto simp add*:*card-mono*)
with *c* **have** *card* (*set* (*g-V G*)) > *1* **by** *simp*
then show *?thesis* **using** *card-length*[*of g-V G*] **by** *auto*
qed

lemma *verts-ge-Suc0* : ¬ [*0*..<*length* (*g-V G*)] = [*0*]
proof
assume [*0*..<*length* (*g-V G*)] = [*0*]
then have *length* [*0*..<*length* (*g-V G*)] = *1* **by** *simp*
with *len-verts-gt1* **show** *False* **by** *auto*
qed

lemma *distinct-verts1*: *distinct* [*0*..<*length* (*g-V G*)]
by *simp*

lemma *distinct-verts2*: *distinct* (*g-V G*)
by (*insert distinct-verts1 verts*) *simp*

lemma *single-entry*: *is-singleton* (*g-V0 G*)
by (*simp add*:*entry0*)

lemma *entry-is-0*: *the-elem* (*g-V0 G*) = *0*
by (*simp add*: *entry0*)

lemma *wf-digraph*: *cfg-doms G* **by** *intro-locales*

lemma *path-entry-prepend-conv*: *path-entry* (*g-E G*) *p n* ⟹ *p* ≠ [] ⟹ ∃ *v*. *path-entry* (*g-E G*) (*tl p*) *v* ∧ (*v*, *n*) ∈ (*g-E G*)
proof (*induct rule*:*path-entry.induct*)
case *path-entry0* **then show** *?case* **by** *auto*
next
case (*path-entry-prepend u v l*)
then show *?case* **by** *auto*
qed

```
lemma path-verts: path-entry (g-E G) p n ⟹ n ∈ set (g-V G)
proof (cases p = [])
  case True
  assume path-entry (g-E G) p n and p = []
  then show ?thesis by (simp add:path-entry0-empty-conv zero-in-verts)
next
  case False
  assume path-entry (g-E G) p n and p ≠ []
  then have (0,n)∈(g-E G)⁺ by (auto simp add:path-entry-is-trancl)
  then show ?thesis using head-is-vert2 by simp
qed

lemma path-in-verts:
  assumes path-entry (g-E G) l v
    shows set l ⊆ set (g-V G)
  using assms
proof (induct rule:path-entry.induct)
  case path-entry0 then show ?case by auto
next
  case (path-entry-prepend u v l)
  then show ?case using path-verts by auto
qed

lemma any-node-exits-path:
  assumes v ∈ set (g-V G)
    shows ∃ p. path-entry (g-E G) p v
  using assms
proof (cases v = 0)
  assume v ∈ set (g-V G) and v = 0
  have path-entry (g-E G) [] 0 by (auto simp add:path-entry0)
  then show ?thesis using ‹v = 0› by auto
next
  assume v ∈ set (g-V G) and v ≠ 0
  with reachable have v ∈ (g-E G)* `` {0} by auto
  then have (0,v) ∈ (g-E G)*  by (simp add:Image-iff)
  then show ?thesis by (auto intro:entry-rtrancl-is-path)
qed

lemma entry0-path: path-entry (g-E G) [] 0
  by (auto simp add:path-entry.path-entry0)

definition reachable :: node ⇒ bool
  where reachable v ≡ v ∈ (g-E G)* `` {0}

lemma path-entry-reachable:
  assumes path-entry (g-E G) p n
    shows reachable n
  using assms reachable
```

```
  by (fastforce intro:path-verts simp add:reachable-def)

lemma nin-nodes-reachable: n ∉ set (g-V G) ⟹ ¬ reachable n
proof(rule ccontr)
  assume n ∉ set (g-V G) and nn: ¬ ¬ reachable n
  from ‹n ∉ set (g-V G)› have n ≠ 0 using verts-set len-verts-gt0 entry0 by
auto
  from nn have reachable n by auto
  then have n ∈ (g-E G)* `` {0} by (simp add: reachable-def)
  then have (0, n) ∈ (g-E G)* by (auto simp add:Image-def)
  with ‹n ≠ 0› have ∃ n'. (0,n') ∈ (g-E G)* ∧ (n', n) ∈ (g-E G) by (auto
intro:rtranclE)
  then obtain n' where (0,n') ∈ (g-E G)* and (n', n) ∈ (g-E G)by auto
  then have n ∈ set (g-V G) using head-is-vert by auto
  with ‹n ∉ set (g-V G)› show False
    by auto
qed

lemma reachable-path-entry: reachable n ⟹ ∃ p. path-entry (g-E G) p n
proof−
  assume reachable n
  then have (0,n) ∈ (g-E G)* by (auto simp add:reachable-def Image-iff)
  then have 0 = n ∨ 0 ≠ n ∧ (0,n) ∈ (g-E G)+ by (auto simp add: rtrancl-eq-or-trancl)
  then show ?thesis
  proof
    assume 0 = n
    have path-entry (g-E G) [] 0 by (simp add:path-entry0)
    with ‹0 = n› show ?thesis by auto
  next
    assume 0 ≠ n ∧ (0,n) ∈ (g-E G)+
    then have (0,n) ∈ (g-E G)+ by (auto simp add:rtranclpD)
    then have n ∈ set (g-V G) by (simp add:head-is-vert2)
    then show ?thesis by (rule any-node-exits-path)
  qed
qed

lemma path-entry-unconc:
  assumes path-entry (g-E G) (la@lb) w
  obtains v where path-entry (g-E G) lb v
  using assms
  apply (induct la@lb w arbitrary:la lb rule: path-entry.induct)
   apply (fastforce intro:path-entry.intros)
  by (auto intro:path-entry.intros iff add: Cons-eq-append-conv)

lemma path-entry-append-conv:
  path-entry (g-E G) (v#l) w ⟷ (path-entry (g-E G) l v ∧ (v,w) ∈ (g-E G))
proof
  assume path-entry (g-E G) (v # l) w
  then show path-entry (g-E G) l v ∧ (v, w) ∈ g-E G
```

by (*auto simp add*:*path-entry-cons-conv*)
next
assume *path-entry* (*g-E G*) *l v* ∧ (*v*, *w*) ∈ *g-E G*
then show *path-entry* (*g-E G*) (*v* # *l*) *w* **by** (*fastforce intro*: *path-entry-append*)
qed

lemma *takeWhileNot-path-entry*:
assumes *path-entry E p x*
and *v* ∈ *set p*
and *takeWhile* ((≠) *v*) (*rev p*) = *c*
shows *path-entry E* (*rev c*) *v*
using *assms*
proof (*induct rule*: *path-entry.induct*)
case *path-entry0*
then show *?case* **by** *auto*
next
case (*path-entry-prepend u va l*)
then show *?case*
proof(*cases v* ∈ *set l*)
case *True* **note** *v-in* = *this*
then have *takeWhile* ((≠) *v*) (*rev* (*u* # *l*)) = *takeWhile* ((≠) *v*) (*rev l*) **by** *auto*
with *path-entry-prepend.prems*(*2*) **have** *takeWhile* ((≠) *v*) (*rev l*) = *c* **by** *simp*
with *v-in* **show** *?thesis* **using** *path-entry-prepend.hyps*(*3*) **by** *auto*
next
case *False* **note** *v-nin* = *this*
with *path-entry-prepend.prems*(*1*) **have** *v-u*: *v* = *u* **by** *auto*
then have *take-eq*: *takeWhile* ((≠) *v*) (*rev* (*u* # *l*)) = *takeWhile* ((≠) *v*) ((*rev l*) @ [*u*]) **by** *auto*
from *v-nin* **have** ⋀*x*. *x* ∈ *set* (*rev l*) ⟹ ((≠) *v*) *x* **by** *auto*
then have *takeWhile* ((≠) *v*) ((*rev l*) @ [*u*]) = (*rev l*) @ *takeWhile* ((≠) *v*) [*u*]
by (*rule takeWhile-append2*) *simp*
with *v-u take-eq* **have** *takeWhile* ((≠) *v*) (*rev* (*u* # *l*)) = (*rev l*) **by** *simp*
then show *?thesis* **using** *path-entry-prepend.prems*(*2*) *path-entry-prepend.hyps*(*2*) *v-u* **by** *auto*
qed
qed

lemma *path-entry-last*: *path-entry* (*g-E G*) *p n* ⟹ *p* ≠ [] ⟹ *last p* = *0*
apply (*induct rule*: *path-entry.induct*)
apply *simp*
apply (*simp add*: *path-entry-cons-conv neq-Nil-conv*)
apply (*auto simp add*:*path-entry0-empty-conv*)
done

lemma *path-entry-loop*:
assumes *n-path*: *path-entry* (*g-E G*) *p n*
and *n*: *n* ∈ *set p*
shows ∃ *p*′. *path-entry* (*g-E G*) *p*′ *n* ∧ *n* ∉ *set p*′

using *assms*
proof −
let *?c* = *takeWhile* ((≠) *n*) (*rev* (*p*))
have ∀ *z* ∈ *set ?c*. *z* ≠ *n* **by** (*auto dest*: *set-takeWhileD*)
then have *n-nin*: *n* ∉ *set* (*rev ?c*) **by** *auto*

from *n-path* **obtain** *v* **where** *path-entry* (*g-E G*) (*p*) *v* **using** *path-entry-prepend-conv* **by** *auto*
with *n* **have** *path-entry* (*g-E G*) (*rev ?c*) *n* **by** (*auto intro*:*takeWhileNot-path-entry*)

with *n-nin* **show** *?thesis* **by** *fastforce*
qed

lemma *path-entry-hd-edge*:
assumes *path-entry* (*g-E G*) *pa p*
and *pa* ≠ []
shows (*hd pa*, *p*) ∈ (*g-E G*)
using *assms*
by (*induct rule*: *path-entry.induct*) *auto*

lemma *path-entry-edge*:
assumes *pa* ≠ []
and *path-entry* (*g-E G*) *pa p*
shows ∃ *u*∈*set pa*. (*path-entry* (*g-E G*) (*rev* (*takeWhile* ((≠) *u*) (*rev pa*))) *u*) ∧ (*u*,*p*) ∈ (*g-E G*)
using *assms*
proof−
from *assms* **have** *1*: *path-entry* (*g-E G*) (*rev* (*takeWhile* ((≠) (*hd pa*)) (*rev pa*))) (*hd pa*) **by** (*auto intro*:*takeWhileNot-path-entry*)
from *assms* **have** *2*: (*hd pa*, *p*)∈ (*g-E G*) **by** (*auto intro*: *path-entry-hd-edge*)
have *hd pa* ∈ *set pa* **using** *assms*(*1*) **by** *auto*
with *1 2* **show** *?thesis* **by** *auto*
qed

definition *is-tail* :: *node* ⇒ *node* × *node* ⇒ *bool*
where *is-tail v e* = (*v* = *tail G e*)

definition *is-head* :: *node* ⇒ *node* × *node* ⇒ *bool*
where *is-head v e* = (*v* = *head G e*)

definition *succs*:: *node* ⇒ *node set*
where *succs v* = (*g-E G*) ‘‘ {*v*}

lemma *succ-in-verts*: *s* ∈ *succs n* ⟹ {*s*,*n*} ⊆ *set* (*g-V G*)
by(*simp add*:*succs-def tail-is-vert head-is-vert*)

lemma *succ0-not-nil*: *succs 0* ≠ {}
using *succ-of-entry0* **by** (*auto simp add*:*succs-def*)

```
definition prevs:: node ⇒ node set where
  prevs v = (converse (g-E G))‘‘ {v}

lemma v ∈ succs u ⟷ u ∈ prevs v
  by (auto simp add:succs-def prevs-def)

lemma succ-edge: ∀ v ∈ succs n. (n,v) ∈ g-E G
  by (auto simp add:succs-def)

lemma prev-edge: u ∈ set (g-V G) ⟹ ∀ v ∈ prevs u. (v, u) ∈ g-E G
  by (auto simp add:prevs-def)

lemma succ-in-G: ∀ v ∈ succs n. v ∈ set (g-V G)
  by (auto simp add: succs-def dest:head-in-verts)

lemma succ-is-subset-of-verts: ∀ v ∈ set (g-V G). succs v ⊆ set(g-V G)
  by (insert succ-in-G) auto

lemma fin-succs: ∀ v ∈ set (g-V G). finite (succs v)
  by (insert succ-is-subset-of-verts) (auto intro:rev-finite-subset)

lemma fin-succs': v < length (g-V G) ⟹ finite (succs v)
  by (subgoal-tac v ∈ set (g-V G))
   (auto simp add: fin-succs verts-set)

lemma succ-range: ∀ v ∈ succs n. v < length (g-V G)
  by (insert succ-in-G verts-set) auto

lemma path-entry-gt:
  assumes ∀ p. path-entry E p n ⟶ d ∈ set p
      and ∀ p. path-entry E p n' ⟶ n ∈ set p
    shows ∀ p. path-entry E p n' ⟶ d ∈ set p
  using assms
proof (intro strip)
  fix p
  let ?npath = takeWhile ((≠) n) (rev p)
  have sub: set ?npath ⊆ set p apply (induct p) by (auto dest:set-takeWhileD)

  assume ass: path-entry E p n'
  with assms(2) have n-in-p: n ∈ set p by auto
  then have n ∈ set (rev p) by auto
  with ass have path-entry E (rev ?npath) n
    using takeWhileNot-path-entry by auto
  with assms(1) have d ∈ set ?npath by fastforce
  with sub show d ∈ set p by auto
qed

definition dominate :: nat ⇒ nat ⇒ bool
  where dominate n1 n2 ≡
```

```
      ∀ pa. path-entry (g-E G) pa n2 ⟶
      (n1 ∈ set pa ∨ n1 = n2)

definition strict-dominate:: nat ⇒ nat ⇒ bool where
  strict-dominate n1 n2 ≡
   ∀ pa. path-entry (g-E G) pa n2 ⟶
   (n1 ∈ set pa ∧ n1 ≠ n2)

lemma any-dominate-unreachable: ¬ reachable n ⟹ dominate d n
proof(unfold reachable-def dominate-def)
  assume ass: n ∉ (g-E G)* `` {0}

  have ¬ (∃ p. path-entry (g-E G) p n)
  proof (rule ccontr)
    assume ¬ (¬ (∃ p. path-entry (g-E G) p n))
    then obtain p where p: path-entry (g-E G) p n by auto
    then have n = 0 ∨ reachable n by (auto intro:path-entry-reachable)
    then show False
    proof
      assume n = 0
      then show False using ass by auto
    next
      assume reachable n
      then show False using ass by (auto simp add:reachable-def)
    qed
  qed
  then show ∀ pa. path-entry (g-E G) pa n ⟶ d ∈ set pa ∨ d = n by auto
qed

lemma any-sdominate-unreachable: ¬ reachable n ⟹ strict-dominate d n
proof(unfold reachable-def strict-dominate-def)
  assume ass:n ∉ (g-E G)* `` {0}

  have ¬ (∃ p. path-entry (g-E G)  p n)
  proof (rule ccontr)
    assume ¬ (¬ (∃ p. path-entry (g-E G) p n))
    then obtain p where p: path-entry (g-E G) p n by auto
    then have n = 0 ∨ reachable n by (auto intro:path-entry-reachable)
    then show False
    proof
      assume n = 0
      then show False using ass by auto
    next
      assume reachable n
      then show False using ass by (auto simp add:reachable-def)
    qed
  qed
  then show ∀ pa. path-entry (g-E G)  pa n ⟶ d ∈ set pa ∧ d ≠ n by auto
qed
```

```
lemma dom-reachable: reachable n ⟹ dominate d n ⟹ reachable d
proof -
  assume reach-n: reachable n
     and dom-n: dominate d n
  from reach-n have ∃ p. path-entry (g-E G)  p n by (rule reachable-path-entry)
  then obtain p where p: path-entry (g-E G) p n by auto

  show reachable d
  proof (cases d ≠ n)
    case True
    with dom-n p have d-in: d ∈ set p by (auto simp add:dominate-def)
    let ?pa = takeWhile ((≠) d) (rev p)
   from d-in p have path-entry (g-E G) (rev ?pa) d using takeWhileNot-path-entry
by auto
    then show ?thesis using path-entry-reachable by auto
  next
    case False
    with reach-n show ?thesis by auto
  qed
qed

lemma dominate-refl: dominate n n
  by (simp add:dominate-def)

lemma entry0-dominates-all: ∀ p ∈ set (g-V G). dominate 0 p
proof(intro strip)
  fix p
  assume p ∈ set (g-V G)
  show dominate 0 p
  proof (cases p = 0)
    case True
    then show ?thesis by (auto simp add:dominate-def)
  next
    case False
    assume p-neq0: p ≠ 0
    have ∀ pa. path-entry (g-E G) pa p ⟶ 0 ∈ set pa
    proof (intro strip)
      fix pa
      assume path-p: path-entry (g-E G) pa p
      show 0 ∈ set pa
      proof (cases pa ≠ [])
        case True note pa-n-nil = this
        with path-p have last-pa: last pa = 0 using path-entry-last by auto
        from pa-n-nil have last pa ∈ set pa by simp
        with last-pa show ?thesis by simp
      next
        case False
        with path-p have p = 0 by (simp add:path-entry0-empty-conv)
```

```
      with p-neq0 show ?thesis by auto
    qed
  qed
  then show ?thesis by (auto simp add:dominate-def)
 qed
qed

lemma strict-dominate i j ⟹ j ∈ set (g-V G) ⟹ i ≠ j
 using any-node-exits-path
 by (auto simp add:strict-dominate-def)

definition non-strict-dominate:: nat ⇒ nat ⇒ bool where
 non-strict-dominate n1 n2 ≡ ∃ pa. path-entry (g-E G) pa n2 ∧ (n1 ∉ set pa)

lemma any-sdominate-0: n ∈ set (g-V G) ⟹ non-strict-dominate n 0
 apply (simp add:non-strict-dominate-def)
 by (auto intro:path-entry0)


lemma non-sdominate-succ: (i,j ) ∈ g-E G ⟹ k ≠ i ⟹ non-strict-dominate k
i ⟹ non-strict-dominate k j
proof −
 assume i-j: (i,j ) ∈ g-E G and k-neq-i: k ≠ i and non-strict-dominate k i
  then obtain pa where path-entry (g-E G) pa i and k-nin-pa: k ∉ set pa by
(auto simp add:non-strict-dominate-def)
 with i-j have path-entry (g-E G) (i#pa) j by (auto simp add:path-entry-prepend)
 with k-neq-i k-nin-pa show ?thesis by (auto simp add:non-strict-dominate-def)
qed

lemma any-node-non-sdom0: non-strict-dominate k 0
 by (auto intro:entry0-path simp add:non-strict-dominate-def)

lemma nonstrict-eq: non-strict-dominate i j ⟹ ¬ strict-dominate i j
 by (auto simp add:non-strict-dominate-def strict-dominate-def)

lemma dominate-trans:
 assumes dominate n1 n2
    and dominate n2 n3
  shows dominate n1 n3
 using assms
proof(cases n1 = n2)
 case True
 then show ?thesis using assms(2) by auto
next
 case False
 then show dominate n1 n3
 proof (cases n1 = n3)
  case True
  then show ?thesis by (auto simp add:dominate-def)
```

next
  case *False*
  show *dominate n1 n3*
  proof (*cases n2 = n3*)
    case *True*
    then show *?thesis* **using** *assms*(*1*) **by** *auto*
  next
    case *False*
    with ‹*n1* ≠ *n2*› ‹*n1* ≠ *n3*› **show** *?thesis*
    proof (*auto simp add*: *dominate-def*)
      fix *pa*
      assume *n1* ≠ *n2* **and** *n1* ≠ *n3* **and** *n2* ≠ *n3*
      from ‹*n1* ≠ *n2*› *assms*(*1*) **have** *n1-n2-pa*: ∀ *pa*. *path-entry* (*g-E G*) *pa n2* ⟶ *n1* ∈ *set pa*
        by (*auto simp add*:*dominate-def*)
      from ‹*n2* ≠ *n3*› *assms*(*2*) **have** ∀ *pa*. *path-entry* (*g-E G*) *pa n3* ⟶ *n2* ∈ *set pa*
        by (*auto simp add*:*dominate-def*)
      with *n1-n2-pa* **have** ∀ *pa*. *path-entry* (*g-E G*) *pa n3* ⟶ *n1* ∈ *set pa*
        by (*rule path-entry-gt*)
      then show ⋀*pa*. *path-entry* (*g-E G*) *pa n3* ⟹ *n1* ∈ *set pa* **by** *auto*
    qed
  qed
qed
qed

lemma *len-takeWhile-lt*: *x* ∈ *set xs* ⟹ *length* (*takeWhile* ((≠) *x*) *xs*) < *length xs*
  by (*induct xs*) *auto*

lemma *len-takeWhile-comp*:
  assumes *n1* ∈ *set xs*
    and *n2* ∈ *set xs*
    and *n1* ≠ *n2*
  shows *length* (*takeWhile* ((≠) *n1*) *xs*) ≠ *length* (*takeWhile* ((≠) *n2*) *xs*)
  using *assms*
  by (*induct xs*) *auto*

lemma *len-takeWhile-comp1*:
  assumes *n1* ∈ *set xs*
    and *n2* ∈ *set xs*
    and *n1* ≠ *n2*
  shows *length* (*takeWhile* ((≠) *n1*) (*rev* (*x* # *xs*))) ≠ *length* (*takeWhile* ((≠) *n2*) (*rev* (*x* # *xs*)))
  using *assms len-takeWhile-comp*[*of n1 rev xs n2*] **by** *fastforce*

lemma *len-takeWhile-comp2*:
  assumes *n1* ∈ *set xs*
    and *n2* ∉ *set xs*
  shows *length* (*takeWhile* ((≠) *n1*) (*rev* (*x* # *xs*))) ≠ *length* (*takeWhile* ((≠)

n2) (*rev* (*x* # *xs*)))
  using *assms*
proof−
  let *?xs1* = *takeWhile* ((≠) *n1*) (*rev* (*x* # *xs*))
  let *?xs2* = *takeWhile* ((≠) *n2*) (*rev* (*x* # *xs*))
  from *assms* **have** *len1*: *length* (*takeWhile* ((≠) *n1*) (*rev xs*)) < *length* (*rev xs*)
    using *len-takeWhile-lt*[*of -rev xs*] **by** *auto*

  from *assms*(*1*) **have** *?xs1* = *takeWhile* ((≠) *n1*) (*rev xs*) **by** *auto*
  then have *len2*: *length ?xs1* < *length* (*rev xs*) **using** *len1* **by** *auto*

  from *assms*(*2*) **have** *takeWhile* ((≠) *n2*) (*rev xs* @ [*x*]) = (*rev xs*) @ *takeWhile* ((≠) *n2*) [*x*]
    by (*fastforce intro*:*takeWhile-append2*)
  then have *?xs2* = (*rev xs*) @ *takeWhile* ((≠) *n2*) [*x*] **by** *simp*
  then show *?thesis* **using** *len2* **by** *auto*
qed

lemma *len-compare1*:
  assumes *n1* = *x* **and** *n2* ≠ *x*
    shows *length* (*takeWhile* ((≠) *n1*) (*rev* (*x* # *xs*))) ≠ *length* (*takeWhile* ((≠) *n2*) (*rev* (*x* # *xs*)))
  using *assms*
proof(*cases n1* ∈ *set xs* ∧ *n2* ∈ *set xs*)
  case *True*
  with *assms* **show** *?thesis* **using** *len-takeWhile-comp1* **by** *fastforce*
next
  let *?xs1* = *takeWhile* ((≠) *n1*) (*rev* (*x* # *xs*))
  let *?xs2* = *takeWhile* ((≠) *n2*) (*rev* (*x* # *xs*))

  case *False*
  then have *n1* ∈ *set xs* ∧ *n2* ∉ *set xs* ∨ *n1* ∉ *set xs* ∧ *n2* ∈ *set xs* ∨ *n1* ∉ *set xs* ∧ *n2* ∉ *set xs* **by** *auto*
  then show *?thesis*
  proof
    assume *n1* ∈ *set xs* ∧ *n2* ∉ *set xs*
    then show *?thesis* **by** (*fastforce dest*: *len-takeWhile-comp2*)
  next
    assume *n1* ∉ *set xs* ∧ *n2* ∈ *set xs* ∨ *n1* ∉ *set xs* ∧ *n2* ∉ *set xs*
    then show *?thesis*
    proof
      assume *n1* ∉ *set xs* ∧ *n2* ∈ *set xs*
      then have *n1*: *n1* ∉ *set xs* **and** *n2*: *n2* ∈ *set xs* **by** *auto*
      have *length ?xs2* ≠ *length ?xs1* **using** *len-takeWhile-comp2*[*OF n2 n1*] **by** *auto*
      then show *?thesis* **by** *simp*
    next
      assume *n1* ∉ *set xs* ∧ *n2* ∉ *set xs*
      then have *n1-nin*: *n1* ∉ *set xs* **and** *n2-nin*: *n2* ∉ *set xs* **by** *auto*

```
      then  have t1: takeWhile ((≠) n1) (rev xs @ [x]) = (rev xs) @ takeWhile ((≠) n1) [x]
          and    takeWhile ((≠) n2) (rev xs @ [x]) = (rev xs) @ takeWhile  ((≠) n2) [x]
       by (fastforce intro:takeWhile-append2)+
     with ‹n1 = x› ‹n2 ≠ x›  have t1': takeWhile ((≠) n1) (rev xs @ [x]) = rev xs
                       and     takeWhile ((≠) n2) (rev xs @ [x]) = (rev xs) @ [x] by auto
      then have length (takeWhile ((≠) n2) (rev xs @ [x])) = length ((rev xs) @ [x])
       using arg-cong[of takeWhile ((≠) n2) (rev xs @ [x]) rev xs @ [x] length] by fastforce
     with t1' show ?thesis by auto
   qed
 qed
qed

lemma len-compare2:
 assumes n1 ∈ set xs
     and n1 ≠ n2
    shows length (takeWhile ((≠) n1) (rev (x # xs))) ≠ length (takeWhile ((≠) n2) (rev (x # xs)))
 using assms
 apply(case-tac n2 ∈ set xs)
  apply (fastforce dest: len-takeWhile-comp1 )
 apply (fastforce dest:len-takeWhile-comp2)
 done

lemma len-takeWhile-set:
 assumes length (takeWhile ((≠) n1) xs) > length (takeWhile ((≠) n2) xs)
     and n1 ≠ n2
     and n1 ∈ set xs
     and n2 ∈ set xs
   shows set (takeWhile ((≠) n2) xs)  ⊆ set (takeWhile ((≠) n1) xs)
 using assms
proof (induct xs)
 case Nil then show ?case by auto
next
 case (Cons y ys)
 note ind-hyp = Cons(1)
 note len-n2-lt-n1-y-ys = Cons(2)
 note n1-n-n2 = Cons(3)
 note n1-in-y-ys = Cons(4)
 note n2-in-y-ys = Cons(5)

 let ?ys1-take = takeWhile ((≠) n1) ys
 let ?ys2-take = takeWhile ((≠) n2) ys
```

```
  show ?case
  proof(cases n1 ∈ set ys)
    case True note n1-in-ys = this
    show ?thesis
    proof(cases n2 ∈ set ys)
      case True note n2-in-ys = this
      show ?thesis
      proof (cases n1 ≠ y)
        case True note n1-neq-y = this
        show ?thesis
        proof (cases n2 ≠ y)
          case True note n2-neq-y = this
          from len-n2-lt-n1-y-ys have length ?ys2-take < length ?ys1-take
            using n1-n-n2 n1-in-ys n2-in-ys n1-neq-y n2-neq-y by (induct ys) auto
          from ind-hyp[OF this n1-n-n2 n1-in-ys n2-in-ys]
          have set (takeWhile ((≠) n2) ys) ⊆ set (takeWhile ((≠) n1) ys) by auto
          then show ?thesis using n1-neq-y n2-neq-y by (induct ys) auto
        next
          case False
          with n1-n-n2 show ?thesis by auto
        qed
      next
        case False
        with n1-n-n2 show ?thesis using len-n2-lt-n1-y-ys by auto
      qed
    next
      case False
      with n2-in-y-ys have n2 = y by auto
      then show ?thesis by auto
    qed
  next
    case False
    with n1-in-y-ys have n1 = y by auto
    with n1-n-n2 show ?thesis using len-n2-lt-n1-y-ys by auto
  qed
qed

lemma reachable-dom-acyclic:
  assumes reachable n2
      and dominate n1 n2
      and dominate n2 n1
    shows n1 = n2
  using assms
proof -
  from assms(1) assms(2) have reachable n1 by (auto intro: dom-reachable)
  then have ∃ pa. path-entry (g-E G) pa n1 by (auto intro: reachable-path-entry)
  then obtain pa where pa: path-entry (g-E G) pa n1 by auto

  let ?n-take-n1 = takeWhile ((≠) n1) (rev pa)
```

let *?n-take-n2* = *takeWhile* ((≠) *n2*) (*rev pa*)

show *n1* = *n2*
proof(*rule ccontr*)
assume *n1-neq-n2*: *n1* ≠ *n2*
then have *pa-n1-n2*: ∀ *pa*. *path-entry* (*g-E G*) *pa n2* ⟶ *n1* ∈ *set pa*
and *pa-n2-n1*: ∀ *pa*. *path-entry* (*g-E G*) *pa n1* ⟶ *n2* ∈ *set pa* **using** *assms*(*2*) *assms*(*3*)
by (*auto simp add*:*dominate-def*)
then have *n1-n1-pa*: ∀ *pa*. *path-entry* (*g-E G*) *pa n1* ⟶ *n1* ∈ *set pa* **by** (*rule path-entry-gt*)
with *pa pa-n2-n1* **have** *n1-in-pa*: *n1* ∈ *set pa*
and *n2-in-pa*: *n2* ∈ *set pa* **by** *auto*
with *n1-neq-n2* **have** *len-neq*: *length ?n-take-n1* ≠ *length ?n-take-n2*
by (*auto simp add*: *len-takeWhile-comp*)

from *pa n1-in-pa n2-in-pa* **have** *path1*: *path-entry* (*g-E G*) (*rev ?n-take-n1*) *n1*

and *path2*: *path-entry* (*g-E G*) (*rev ?n-take-n2*) *n2*
using *takeWhileNot-path-entry* **by** *auto*

have *n1-not-in*: *n1* ∉ *set ?n-take-n1* **by** (*auto dest*: *set-takeWhileD*[*of* - - *rev pa*])
have *n2-not-in*: *n2* ∉ *set ?n-take-n2* **by** (*auto dest*: *set-takeWhileD*[*of* - - *rev pa*])

show *False*
proof(*cases length ?n-take-n1* > *length ?n-take-n2*)
case *True*
then have *set ?n-take-n2* ⊆ *set ?n-take-n1*
using *n1-in-pa n2-in-pa* **by** (*auto dest*: *len-takeWhile-set*[*of* - *rev pa*])
then have *n1* ∉ *set ?n-take-n2* **using** *n1-not-in* **by** *auto*
with *path2* **show** *False* **using** *pa-n1-n2* **by** *auto*
next
case *False*
with *len-neq* **have** *length ?n-take-n2* > *length ?n-take-n1* **by** *auto*
then have *set ?n-take-n1* ⊆ *set ?n-take-n2*
using *n1-neq-n2 n2-in-pa n1-in-pa* **by** (*auto dest*: *len-takeWhile-set*)
then have *n2* ∉ *set ?n-take-n1* **using** *n2-not-in* **by** *auto*
with *path1* **show** *False* **using** *pa-n2-n1* **by** *auto*
qed
qed
qed

lemma *sdom-dom*: *strict-dominate n1 n2* ⟹ *dominate n1 n2*
by (*auto simp add*:*strict-dominate-def dominate-def*)

lemma *dominate-sdominate*: *dominate n1 n2* ⟹ *n1* ≠ *n2* ⟹ *strict-dominate n1 n2*

by (*auto simp add*:*strict-dominate-def dominate-def*)

lemma *sdom-neq*:
  assumes *reachable n2*
      and *strict-dominate n1 n2*
    shows *n1* ≠ *n2*
  using *assms*
proof –
  from *assms*(*1*) **have** ∃ *p*. *path-entry* (*g-E G*) *p n2* **by** (*rule reachable-path-entry*)

  then obtain *p* **where** *path-entry* (*g-E G*) *p n2* **by** *auto*
  with *assms*(*2*) **show** *?thesis* **by** (*auto simp add*:*strict-dominate-def*)
qed

lemma *reachable-dom-acyclic2*:
  assumes *reachable n2*
      and *strict-dominate n1 n2*
    shows ¬ *dominate n2 n1*
  using *assms*
proof –
  from *assms* **have** *n1-dom-n2*: *dominate n1 n2* **and** *n1-neq-n2*: *n1* ≠ *n2*
    by (*auto simp add*:*sdom-dom sdom-neq*)
  with *assms*(*1*) **have** *dominate n2 n1* ⟹ *n1* = *n2* **using** *reachable-dom-acyclic*
by *auto*
  with *n1-neq-n2* **show** *?thesis* **by** *auto*
qed

lemma *not-dom-eq-not-sdom*: ¬ *dominate n1 n2* ⟹ ¬ *strict-dominate n1 n2*
  by (*auto simp add*:*strict-dominate-def dominate-def*)

lemma *reachable-sdom-acyclic*:
  assumes *reachable n2*
      and *strict-dominate n1 n2*
    shows ¬ *strict-dominate n2 n1*
  using *assms*
  apply (*insert reachable-dom-acyclic2*[*OF assms*(*1*) *assms*(*2*)])
  by (*auto simp add*:*not-dom-eq-not-sdom*)

lemma *strict-dominate-trans1*:
  assumes *strict-dominate n1 n2*
      and *dominate n2 n3*
    shows *strict-dominate n1 n3*
  using *assms*
proof (*cases reachable n2*)
  case *True* **note** *reach-n2* = *this*
  with *assms*(*1*) **have** *n1-dom-n2*: *dominate n1 n2* **and** *n1-neq-n2*: *n1* ≠ *n2*
    by (*auto simp add*:*sdom-dom sdom-neq*)
  with *assms*(*2*) **have** *n1-dom-n3*: *dominate n1 n3* **by** (*auto intro*: *dominate-trans*)
  have *n1-neq-n3*: *n1* ≠ *n3*

```
  proof (rule ccontr)
    assume ¬ n1 ≠ n3 then have n1 = n3 by simp
    with assms(2) have n2-dom-n1: dominate n2 n1 by simp
    with reach-n2 n1-dom-n2 have n1 = n2 by (auto dest:reachable-dom-acyclic)
    with n1-neq-n2 show False by auto
  qed
  with n1-dom-n3 show ?thesis by (simp add:strict-dominate-def dominate-def)
next
  case False note not-reach-n2 = this
  have ¬ reachable n3
  proof (rule ccontr)
    assume ¬ ¬ reachable n3
    with assms(2) have reachable n2 by (auto intro: dom-reachable)
    with not-reach-n2 show False by auto
  qed
  then show ?thesis by (auto intro:any-sdominate-unreachable)
qed

lemma strict-dominate-trans2:
  assumes dominate n1 n2
      and strict-dominate n2 n3
    shows strict-dominate n1 n3
  using assms
proof (cases reachable n3)
  case True
  with assms(2) have n2-dom-n3: dominate n2 n3 and n1-neq-n2: n2 ≠ n3
    by (auto simp add:sdom-dom sdom-neq)
  with assms(1) have n1-dom-n3: dominate n1 n3 by (auto intro: dominate-trans)
  have n1-neq-n3: n1 ≠ n3
  proof (rule ccontr)
    assume ¬ n1 ≠ n3 then have n1 = n3 by simp
    with assms(1) have dominate n3 n2 by simp
    with ‹reachable n3› n2-dom-n3 have n2 = n3 by (auto dest:reachable-dom-acyclic)
    with n1-neq-n2 show False by auto
  qed
  with n1-dom-n3 show ?thesis by (simp add:strict-dominate-def dominate-def)
next
  case False
  then have ¬ reachable n3 by simp
  then show ?thesis by (auto intro:any-sdominate-unreachable)
qed

lemma strict-dominate-trans:
  assumes strict-dominate n1 n2
      and strict-dominate n2 n3
    shows strict-dominate n1 n3
  using assms
  apply(subgoal-tac dominate n2 n3)
   apply(rule strict-dominate-trans1)
```

```
  apply (auto simp add: strict-dominate-def dominate-def)
  done

lemma sdominate-dominate-succs:
  assumes i-sdom-j:    strict-dominate i j
      and j-in-succ-k: j ∈ succs k
    shows              dominate i k
proof (rule ccontr)
  assume ass:¬ dominate i k
  then obtain p where path-k: path-entry (g-E G)  p k and i-nin-p: i ∉ set p by
(auto simp add:dominate-def)
  with j-in-succ-k i-sdom-j have i: i = k ∨ i = j by (auto intro:path-entry-append
simp add:succs-def strict-dominate-def)

  from j-in-succ-k have reachable j using succ-in-verts reachable by (auto simp
add:reachable-def)
  with i-sdom-j have i ≠ j by (auto simp add: sdom-neq)
  with i have i = k by auto
  then have dominate i k by (auto simp add:dominate-refl)
  with ass show False by auto
qed

end

end
```

2 More auxiliary lemmas for Lists Sorted wrt <

```
theory Sorted-Less2
  imports Main HOL−Data-Structures.Cmp HOL−Data-Structures.Sorted-Less
begin

lemma Cons-sorted-less: sorted (rev xs) ⟹ ∀ x∈set xs. x < p ⟹   sorted (rev
(p # xs))
  by (induct xs) (auto simp add:sorted-wrt-append)

lemma Cons-sorted-less-nth:  ∀ x<length xs. xs ! x < p ⟹ sorted (rev xs) ⟹
sorted (rev (p # xs))
  apply(subgoal-tac ∀ x∈set xs. x < p)
  apply(fastforce dest:Cons-sorted-less)
  apply(auto simp add: set-conv-nth)
  done

lemma distinct-sorted-rev: sorted (rev xs) ⟹ distinct xs
  by (induct xs) (auto simp add:sorted-wrt-append)

lemma sorted-le2lt:
  assumes List.sorted xs
      and distinct xs
```

```
    shows sorted xs
  using assms
proof (induction xs)
  case Nil then show ?case by auto
next
  case (Cons x xs)
  note ind-hyp-xs = Cons(1)
  note sorted-le-x-xs = Cons(2)
  note dist-x-xs = Cons(3)
  from dist-x-xs have x-neq-xs: ∀ v ∈ set xs. x ≠ v
                and      dist: distinct xs by auto
  from sorted-le-x-xs have sorted-le-xs: List.sorted xs
                     and       x-le-xs: ∀ v ∈ set xs. v ≥ x by auto
  from x-neq-xs x-le-xs have x-lt-xs: ∀ v ∈ set xs. v > x by fastforce
  from ind-hyp-xs[OF sorted-le-xs dist] have sorted xs by auto
  with x-lt-xs show ?case by auto
qed

lemma sorted-less-sorted-list-of-set: sorted (sorted-list-of-set S)
  by (auto intro:sorted-le2lt)

lemma distinct-sorted: sorted xs ⟹ distinct xs
  by (induct xs) (auto simp add: sorted-wrt-append)

lemma sorted-less-set-unique:
  assumes sorted xs
      and sorted ys
      and set xs = set ys
    shows xs = ys
  using assms
proof −
  from assms(1) have distinct xs and List.sorted xs by (induct xs) auto
  also from assms(2) have distinct ys and List.sorted ys by (induct ys) auto
  ultimately show xs = ys using assms(3) by (auto intro: sorted-distinct-set-unique)
qed

lemma sorted-less-rev-set-unique:
  assumes sorted (rev xs)
      and sorted (rev ys)
      and set xs = set ys
    shows xs = ys
  using assms sorted-less-set-unique[of rev xs rev ys] by auto

lemma sorted-less-set-eq:
  assumes sorted xs
    shows xs = sorted-list-of-set (set xs)
  using assms
  apply(subgoal-tac sorted (sorted-list-of-set (set xs)))
   apply(auto intro: sorted-less-set-unique sorted-le2lt)
```

```
  done

lemma sorted-less-rev-set-eq:
  assumes sorted (rev xs)
    shows sorted-list-of-set (set xs) = rev xs
  using assms sorted-less-set-eq[of rev xs] by auto

lemma sorted-insort-remove1: sorted w ⟹ (insort a (remove1 a w)) = sorted-list-of-set
(insert a (set w))
proof−
  assume sorted w
 then have (sorted-list-of-set (set w − {a})) = remove1 a w using sorted-less-set-eq
    by (fastforce simp add:sorted-list-of-set-remove)
  hence insort a (remove1 a w) = insort a (sorted-list-of-set (set w − {a})) by
simp
  then show ?thesis by (auto simp add:sorted-list-of-set-insert)
qed

end
```

3 Operations on sorted lists

```
theory Sorted-List-Operations2
imports Sorted-Less2
begin
```

The definition and the inter_sorted_correct lemma in this theory are the same as those in Collections [2]. except the former is for a descending list while the latter is for an ascending one.

```
fun inter-sorted-rev :: 'a::{linorder} list ⇒ 'a list ⇒ 'a list where
   inter-sorted-rev [] l2 = []
 | inter-sorted-rev l1 [] = []
 | inter-sorted-rev (x1 # l1) (x2 # l2) =
    (if (x1 > x2) then (inter-sorted-rev l1 (x2 # l2)) else
      (if (x1 = x2) then x1 # (inter-sorted-rev l1 l2) else inter-sorted-rev (x1 #
l1) l2))

lemma inter-sorted-correct :
  assumes l1-OK: sorted (rev l1)
  assumes l2-OK: sorted (rev l2)
    shows sorted (rev (inter-sorted-rev l1 l2)) ∧ set (inter-sorted-rev l1 l2) = set
l1 ∩ set l2
using assms
proof (induct l1 arbitrary: l2)
  case Nil thus ?case by simp
next
  case (Cons x1 l1 l2)
  note x1-l1-props = Cons(2)
```

```
  note l2-props = Cons(3)

  from x1-l1-props have l1-props: sorted (rev l1)
                  and x1-nin-l1: x1 ∉ set l1
                  and x1-gt: ⋀x. x ∈ set l1 ⟹ x1 > x
    by (auto simp add: Ball-def sorted-wrt-append)

  note ind-hyp-l1 = Cons(1)[OF l1-props]
  show ?case
  using l2-props
  proof (induct l2)
    case Nil with x1-l1-props show ?case by simp
  next
    case (Cons x2 l2)
    note x2-l2-props = Cons(2)
    from x2-l2-props have l2-props: sorted (rev l2)
                  and x2-nin-l2: x2 ∉ set l2
                  and x2-gt: ⋀x. x ∈ set l2 ⟹ x2 > x
    by (auto simp  add: Ball-def sorted-wrt-append )

    note ind-hyp-l2 = Cons(1)[OF l2-props]
    show ?case
    proof (cases x1 > x2)
      case True note x1-gt-x2 = this
      have set l1 ∩ set (x2 # l2) = set (x1 # l1)∩ set (x2 # l2)
        using x1-gt-x2 x1-nin-l1 x2-nin-l2 x1-gt x2-gt
        by fastforce
     then show ?thesis using ind-hyp-l1[OF x2-l2-props]  using x1-gt-x2 x1-nin-l1
x2-nin-l2 x1-gt x2-gt
        by (auto simp add:Ball-def sorted-wrt-append)
    next
      case False note x2-ge-x1 = this
      show ?thesis
      proof (cases x1 = x2)
        case True note x1-eq-x2 = this
        then show ?thesis using ind-hyp-l1[OF l2-props]
          using x1-eq-x2  x1-nin-l1 x2-nin-l2 x1-gt x2-gt by (auto simp add:Ball-def
sorted-wrt-append)
      next
        case False note x1-neq-x2 = this
        with x2-ge-x1 have x2-gt-x1 : x2 > x1 by auto
        from ind-hyp-l2 x2-ge-x1 x1-neq-x2 x2-gt x2-nin-l2 x1-gt
        show ?thesis by auto
      qed
    qed
  qed
qed

lemma inter-sorted-rev-refl: inter-sorted-rev xs xs = xs
```

by (*induct xs*) *auto*

lemma *inter-sorted-correct-col*:
assumes *sorted* (*rev xs*)
and *sorted* (*rev ys*)
shows (*inter-sorted-rev xs ys*) = *rev* (*sorted-list-of-set* (*set xs* ∩ *set ys*))
using *assms*
proof−
from *assms* **have** *1*: *sorted* (*rev* (*inter-sorted-rev xs ys*))
and *2*: *set* (*inter-sorted-rev xs ys*) = *set xs* ∩ *set ys* **using** *inter-sorted-correct* **by** *auto*
have *sorted* (*rev* (*rev* (*sorted-list-of-set* (*set xs* ∩ *set ys*)))) **by** (*simp add*:*sorted-less-sorted-list-of-set*)
with *1 2* **show** *?thesis* **by** (*auto intro*:*sorted-less-rev-set-unique*)
qed

lemma *cons-set-eq*: *set* (*x* # *xs*) ∩ *set xs* = *set xs*
by *auto*

lemma *inter-sorted-cons*: *sorted* (*rev* (*x* # *xs*)) ⟹ *inter-sorted-rev* (*x* # *xs*) *xs* = *xs*
proof−
assume *ass*: *sorted* (*rev* (*x* # *xs*))
then have *sorted-xs*: *sorted* (*rev xs*) **by** (*auto simp add*:*sorted-wrt-append*)
with *ass* **have** *inter-sorted-rev* (*x* # *xs*) *xs* = *rev* (*sorted-list-of-set* (*set* (*x* # *xs*) ∩ *set xs*))
by (*simp add*:*inter-sorted-correct-col*)
then have *inter-sorted-rev* (*x* # *xs*) *xs* = *rev* (*rev xs*)**using** *sorted-xs* **by** (*simp only*:*cons-set-eq sorted-less-rev-set-eq*)
then show *?thesis* **using** *sorted-xs* **by** *auto*
qed

end

4 A semilattice of reversed-ordered list

theory *Dom-Semi-List*
imports *Main Jinja.Semilat Sorted-List-Operations2 Sorted-Less2 Cfg*
begin

type-synonym *node* = *nat*

context *cfg-doms*
begin

definition *nodes* :: *nat list*
where *nodes* ≡ (*g-V G*)

definition *nodes-le* :: *node list* ⇒ *node list* ⇒ *bool* **where**
nodes-le xs ys ≡ (*sorted* (*rev ys*) ∧ *sorted* (*rev xs*) ∧ (*set ys*) ⊆ (*set xs*)) ∨ *xs* = *ys*

definition *nodes-sup* ::*node list* ⇒ *node list* ⇒*node list* **where**
nodes-sup = (λ*x y*. *inter-sorted-rev x y*)

definition *nodes-semi* :: *node list sl* **where**
nodes-semi ≡ ((*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set* (*nodes*))), *nodes-le*, *nodes-sup*
)

lemma *subset-nodes-inpow*:
  assumes *sorted* (*rev xs*)
      and *set xs* ⊆ *set nodes*
    shows *xs* ∈ (*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set nodes*))
proof −
 from *assms*(*1*) **have** (*sorted-list-of-set* (*set xs*)) = *rev xs* **by** (*auto intro*:*sorted-less-rev-set-eq*)
  then have *rev* (*rev xs*) = *rev* (*sorted-list-of-set* (*set xs*)) **by** *simp*
  with *assms*(*2*) **show** *?thesis* **by** *auto*
qed

lemma *nil-in-A*: [] ∈ (*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set nodes*))
proof(*simp add*: *Pow-def image-def*)
  have *sorted-list-of-set* {} = [] **by** *auto*
  then show ∃ *x*⊆*set nodes*. *sorted-list-of-set x* = [] **by** *blast*
qed

lemma *single-n-in-A*: *p* < *length nodes* ⟹ [*p*] ∈ (*rev* ∘ *sorted-list-of-set*) ‘ (*Pow*
(*set nodes*))
proof (*unfold nodes-def*)
  let *?S* = (*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set* (*g-V G*)))
  assume *p* < *length* (*g-V G*)
  then have *p*: {*p*} ∈ *Pow* (*set* (*g-V G*)) **by** (*auto simp add*:*Pow-def verts-set*)
  then have [*p*] ∈*?S* **by** (*unfold image-def*) *force*
  then show [*p*] ∈ *?S* **by** *auto*
qed

lemma *inpow-subset-nodes*:
  assumes *xs* ∈ (*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set nodes*))
    shows *set xs* ⊆ *set nodes*
proof −
 from *assms* **obtain** *x* **where** *x*: *x* ∈ *Pow* (*set nodes*) **and** *xs* = (*rev* ∘ *sorted-list-of-set*)
x **by** *auto*
  then have *eq*: *set xs* = *set* (*sorted-list-of-set x*) **by** *auto*
  have ∀ *x* ∈ *Pow* (*set nodes*). *finite x* **by** (*auto intro*: *rev-finite-subset*)
  with *x eq* **show** *set xs* ⊆ *set nodes* **by** *auto*
qed

lemma *inter-in-pow-nodes*:
  assumes *xs* ∈ (*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set nodes*))
    shows (*rev* ∘ *sorted-list-of-set*)(*set xs* ∩ *set ys*) ∈ (*rev* ∘ (*sorted-list-of-set*)) ‘
(*Pow* (*set nodes*))

```
  using assms
proof -
  let ?res = set xs ∩ set ys
  from assms have set xs ⊆ set nodes using inpow-subset-nodes by auto
  then have ?res ⊆ set nodes by auto
  then show ?thesis using subset-nodes-inpow by auto
qed


lemma nodes-le-order: order nodes-le ((rev ∘ sorted-list-of-set) ‘ (Pow (set nodes)))
proof -
  let ?A = (rev ∘ sorted-list-of-set) ‘ (Pow (set nodes))

  have ∀ x ∈ ?A. sorted (rev x) by (auto intro: sorted-less-sorted-list-of-set)
  then have ∀ x∈?A. nodes-le x x by (auto simp add:nodes-le-def)

  moreover have ∀ x∈?A. ∀ y∈?A. (nodes-le x y ∧ nodes-le y x ⟶ x = y)
  proof (intro strip)
    fix x y
    assume x ∈ ?A and y ∈ ?A and nodes-le x y ∧ nodes-le y x
    then have sorted (rev x) ∧ sorted (rev (y::nat list)) ∧  set x = set y ∨ x = y
     by (auto simp add: nodes-le-def intro:subset-antisym sorted-less-sorted-list-of-set)
   then show x = y by (auto dest: sorted-less-rev-set-unique)
 qed

  moreover have ∀ x∈ ?A.   ∀ y∈ ?A. ∀ z∈ ?A . nodes-le x y ∧ nodes-le y  z ⟶
nodes-le x z
    by (auto simp add: nodes-le-def)

  ultimately show ?thesis by (unfold order-def lesub-def lesssub-def) fastforce
qed

lemma nodes-semi-auxi:
  let A = (rev ∘ sorted-list-of-set) ‘ (Pow (set (nodes)));
       r = nodes-le;
       f = (λx y.  (inter-sorted-rev x y))
    in semilat(A, r, f)
proof -
  let ?A = (rev ∘ sorted-list-of-set) ‘ (Pow (set (nodes)))
  let ?r = nodes-le
  let ?f = (λx y. (inter-sorted-rev x y))

  have order ?r ?A by (rule nodes-le-order)

  moreover have closed ?A ?f
  proof (unfold closed-def, intro strip)
    fix xs ys assume xs-in: xs ∈ ?A and ys-in: ys ∈ ?A
    then have sorted-xs: sorted (rev xs)
          and sorted-ys: sorted (rev ys)
```

by (*auto intro*: *sorted-less-sorted-list-of-set*)
then have *inter-xs-ys*: *set* (*?f xs ys*) = *set xs* ∩ *set ys* **and**
sorted-res: *sorted* (*rev* (*?f xs ys*))
using *inter-sorted-correct* **by** *auto*

from *xs-in* **have** *set xs* ⊆ *set nodes* **using** *inpow-subset-nodes* **by** *auto*
with *inter-xs-ys* **have** *set* (*?f xs ys*) ⊆ *set nodes* **by** *auto*
with *sorted-res* **show** $xs \sqcup_{?f} ys \in\ ?A$ **using** *subset-nodes-inpow* **by** (*auto simp add*:*plussub-def*)
qed

moreover have $(\forall x\in ?A.\ \forall y\in ?A.\ x \sqsubseteq_{?r} x \sqcup_{?f} y) \wedge (\forall x\in ?A.\ \forall y\in ?A.\ y \sqsubseteq_{?r} x \sqcup_{?f} y)$
proof(*rule conjI*, *intro strip*)
fix *xs ys*
assume *xs-in*: *xs* ∈ *?A* **and** *ys-in*: *ys* ∈ *?A*
then have *sorted-xs*: *sorted* (*rev xs*) **and** *sorted-ys*: *sorted* (*rev ys*)
by (*auto intro*: *sorted-less-sorted-list-of-set*)
then have *set* (*?f xs ys*) ⊆ *set xs* **and** *sorted-f-xs-ys*: *sorted* (*rev* (*?f xs ys*))
by (*auto simp add*: *inter-sorted-correct*)
then show $xs \sqsubseteq_{?r} xs \sqcup_{?f} ys$ **by** (*simp add*: *lesub-def sorted-xs sorted-ys sorted-f-xs-ys nodes-le-def plussub-def*)
next
show $\forall x\in ?A.\ \forall y\in ?A.\ y \sqsubseteq_{?r} x \sqcup_{?f} y$
proof (*intro strip*)
fix *xs ys*
assume *xs-in*: *xs* ∈ *?A* **and** *ys-in*: *ys* ∈ *?A*
then have *sorted-xs*: *sorted* (*rev xs*) **and** *sorted-ys*: *sorted* (*rev ys*)
by (*auto intro*: *sorted-less-sorted-list-of-set*)
then have *set* (*?f xs ys*) ⊆ *set ys* **and** *sorted-f-xs-ys*: *sorted* (*rev* (*?f xs ys*)) **by** (*auto simp add*: *inter-sorted-correct*)
then show $ys \sqsubseteq_{?r} xs \sqcup_{?f} ys$ **by** (*simp add*: *lesub-def sorted-ys sorted-xs sorted-f-xs-ys nodes-le-def plussub-def*)
qed
qed

moreover have $\forall x\in ?A.\ \forall y\in ?A.\ \forall z\in ?A.\ x \sqsubseteq_{?r} z \wedge y \sqsubseteq_{?r} z \longrightarrow x \sqcup_{?f} y \sqsubseteq_{?r} z$

proof (*intro strip*)
fix *xs ys zs*
assume *xin*: *xs* ∈ *?A* **and** *yin*: *ys* ∈ *?A* **and** *zin*: *zs* ∈ *?A* **and** $xs \sqsubseteq_{?r} zs \wedge ys \sqsubseteq_{?r} zs$
then have *xs-zs*: $xs \sqsubseteq_{?r} zs$ **and** *ys-zs*: $ys \sqsubseteq_{?r} zs$ **and** *sorted-xs*: *sorted* (*rev xs*) **and** *sorted-ys*: *sorted* (*rev ys*) **by** (*auto simp add*: *sorted-less-sorted-list-of-set*)
then have *inter-xs-ys*: *set* (*?f xs ys*) = (*set xs* ∩ *set ys*) **and** *sorted-f-xs-ys*: *sorted* (*rev* (*?f xs ys*))
by (*auto simp add*: *inter-sorted-correct*)

from *xs-zs ys-zs sorted-xs* **have** *sorted-zs*: *sorted* (*rev zs*)

```
and set zs ⊆ set xs
          and set zs ⊆ set ys by (auto simp add: lesub-def
nodes-le-def)
    then have zs: set zs ⊆ set xs ∩ set ys by auto
    with inter-xs-ys sorted-zs sorted-f-xs-ys show xs ⊔?f ys ⊑?r zs
      by (auto simp add:plussub-def lesub-def  sorted-xs sorted-ys sorted-f-xs-ys
sorted-zs nodes-le-def)
  qed
  ultimately show ?thesis  by (unfold semilat-def) simp
qed

lemma nodes-semi-is-semilat: semilat (nodes-semi)
  using nodes-semi-auxi
  by (auto simp add: nodes-sup-def nodes-semi-def)

lemma sorted-rev-subset-len-lt:
  assumes sorted (rev a)
      and sorted (rev b)
      and set a ⊂ set b
    shows length a < length b
  using assms
proof−
  from assms(1) assms(2) have dist-a: distinct a and dist-b: distinct b by (auto
dest: distinct-sorted-rev)
 from assms(3)  have card (set a) < card (set b) by (auto intro: psubset-card-mono)
  with dist-a dist-b show ?thesis by (auto simp add: distinct-card)
qed

lemma wf-nodes-le-auxi: wf {(y, x). (sorted (rev y) ∧ sorted (rev x) ∧ set y ⊂ set
x) ∧ x ≠ y}
  apply(rule wf-measure [THEN wf-subset])
  apply(simp only: measure-def inv-image-def)
  apply clarify
  apply(frule sorted-rev-subset-len-lt)
    defer
    defer
    apply fastforce
  by (auto intro:sorted-less-rev-set-unique)

lemma wf-nodes-le-auxi2:
  wf {(y, x). sorted (rev y) ∧ sorted (rev x) ∧ set y ⊂ set x ∧ rev x ≠ rev y}
  using wf-nodes-le-auxi by auto

lemma  wf-nodes-le: wf {(y, x). nodes-le x y ∧ x ≠ y}
proof −
  have eq-set: {(y, x). (sorted (rev y) ∧ sorted (rev x) ∧ set y ⊆ set x) ∧ x ≠ y}
=
               {(y, x). nodes-le x y ∧ x ≠ y} by (unfold nodes-le-def) auto
  have {(y, x). (sorted (rev y) ∧ sorted (rev x) ∧ set y ⊂ set x) ∧ x ≠ y} =
```

{(*y*, *x*). (*sorted* (*rev y*) ∧ *sorted* (*rev x*) ∧ *set y* ⊆ *set x*) ∧ *x* ≠ *y*}
by (*auto simp add:sorted-less-rev-set-unique*)
from *this wf-nodes-le-auxi* **have** *wf* {(*y*, *x*). (*sorted* (*rev y*) ∧ *sorted* (*rev x*) ∧ *set y* ⊆ *set x*) ∧ *x* ≠ *y*} **by** (*rule subst*)
with *eq-set* **show** *?thesis* **by** (*rule subst*)
qed

lemma *acc-nodes-le*: *acc nodes-le*
apply (*unfold acc-def lesssub-def lesub-def*)
apply (*rule wf-nodes-le*)
done

lemma *acc-nodes-le2*: *acc* (*fst* (*snd nodes-semi*))
apply (*unfold nodes-semi-def*)
apply (*auto simp add*: *lesssub-def lesub-def intro*: *acc-nodes-le*)
done

lemma *nodes-le-refl* [*iff*] : *nodes-le s s*
apply (*unfold nodes-le-def lesssub-def lesub-def*)
apply (*auto*)
done

end

end

5 A kildall's algorithm for computing dominators

theory *Dom-Kildall*
imports *Dom-Semi-List HOL−Library.While-Combinator Jinja.SemilatAlg*
begin

A kildall's algorithm for computing dominators. It uses the ideas and the framework of kildall's algorithm implemented in Jinja [3], and modifications are needed to make it work for a fast algorithm for computing dominators

type-synonym *state-dom* = *nat list*

primrec *propa* ::
$'s\ binop \Rightarrow (nat \times 's)\ list \Rightarrow 's\ list \Rightarrow nat\ list \Rightarrow 's\ list * nat\ list$
where
propa f [] τs *wl* = (τs,*wl*)
| *propa f* (q'# *qs*) τs *wl* = (*let* (q,τ) = q';
u = ($\tau \sqcup_f \tau s!q$);
wl' = (*if* $u = \tau s!q$ *then wl*
else (*insort q* (*remove1 q wl*)))
in propa f qs ($\tau s[q := u]$) wl')

definition *iter* ::

's binop ⇒ 's step-type ⇒ 's list ⇒ nat list ⇒ 's list × nat list
where
iter f step τs w =
while (λ(τs,w). w ≠ [])
(λ(τs,w). let p = hd w
in propa f (step p (τs!p)) τs (tl w))
(τs,w)

definition *unstables :: state-dom ord ⇒ state-dom step-type ⇒ state-dom list ⇒ nat list*
where
unstables r step τs = sorted-list-of-set {p. p < size τs ∧ ¬ stable r step τs p}

definition *kildall :: state-dom ord ⇒state-dom binop ⇒ state-dom step-type ⇒ state-dom list ⇒ state-dom list* **where**
kildall r f step τs = fst(iter f step τs (unstables r step τs))

lemma *init-worklist-is-sorted: sorted (unstables r step τs)*
by *(simp add:sorted-less-sorted-list-of-set unstables-def)*

context *cfg-doms*

begin

definition *transf :: node ⇒ state-dom ⇒ state-dom* **where**
transf n input ≡ (n # input)

definition *exec :: node ⇒ state-dom ⇒ (node × state-dom) list*
where *exec n xs = map (λpc. (pc, (transf n xs))) (rev (sorted-list-of-set(succs n)))*

lemma *transf-res-is-rev: sorted (rev ns) ⟹ n > hd ns ⟹ sorted (rev ((transf n (ns))))*
by *(induct ns) (auto simp add:transf-def sorted-wrt-append)*

abbreviation *start ≡ [] # (replicate (length (g-V G) − 1) ((rev[0..<length(g-V G)])))*

definition *dom-kildall :: state-dom list ⇒ state-dom list*
where *dom-kildall = kildall (fst (snd nodes-semi)) (snd (snd nodes-semi)) exec*

definition *dom:: nat ⇒ nat ⇒ bool* **where**
dom i j ≡(let res = (dom-kildall start) !j in i ∈ (set res) ∨ i = j)

lemma *dom-refl: dom i i*
by *(unfold dom-def) simp*

definition *strict-dom :: nat ⇒ nat ⇒ bool* **where**

strict-dom i j ≡ (*let res* = (*dom-kildall start*) !*j in* *i* ∈ *set res*)

lemma *strict-domI1*: (*dom-kildall* ([] # (*replicate* (*length* (*g-V G*) − *1*) ((*rev*[*0*..<*length*(*g-V G*)])))))!*j* = *res* ⟹ *i* ∈ *set res* ⟹ *strict-dom i j*
by (*auto simp add*:*strict-dom-def*)

lemma *strict-domD*:
strict-dom i j ⟹
dom-kildall (([] # (*replicate* (*length* (*g-V G*) − *1*) ((*rev*[*0*..<*length*(*g-V G*)])))))!*j*
= *a* ⟹
i ∈ *set a*
by (*auto simp add*:*strict-dom-def*)

lemma *sdom-dom*: *strict-dom i j* ⟹ *dom i j*
by (*unfold strict-dom-def*) (*auto simp add*:*dom-def*)

lemma *not-sdom-not-dom*: ¬*strict-dom i j* ⟹ *i* ≠ *j* ⟹ ¬*dom i j*
by (*unfold strict-dom-def*) (*auto simp add*:*dom-def*)

lemma *dom-sdom*: *dom i j* ⟹ *i* ≠ *j* ⟹ *strict-dom i j*
by (*unfold dom-def*) (*auto simp add*:*dom-def strict-dom-def*)

end

end

6 Properties of the kildall's algorithm on the semi-lattice

theory *Dom-Kildall-Property*
imports *Dom-Kildall Jinja.Listn Jinja.Kildall-1*
begin

lemma *sorted-list-len-lt*: *x* ⊂ *y* ⟹ *finite y* ⟹ *length* (*sorted-list-of-set x*) < *length* (*sorted-list-of-set y*)
proof−
let *?x* = *sorted-list-of-set x*
let *?y* = *sorted-list-of-set y*
assume *x-y*: *x* ⊂ *y* **and** *fin-y*: *finite y*
then have *card-x-y*: *card x* < *card y* **and** *fin-x*: *finite x*
by (*auto simp add*:*psubset-card-mono finite-subset*)
with *fin-y* **have** *length ?x* = *card x* **and** *length ?y* = *card y* **by** *auto*
with *card-x-y* **show** *?thesis* **by** *auto*
qed

lemma *wf-sorted-list*:

wf $((\lambda(x,y).\ (sorted\text{-}list\text{-}of\text{-}set\ x,\ sorted\text{-}list\text{-}of\text{-}set\ y))\ \text{`}\ finite\text{-}psubset)$
apply (*unfold finite-psubset-def*)
apply (*rule wf-measure* [*THEN wf-subset*])
apply (*simp add*: *measure-def inv-image-def image-def*)
by (*auto intro*: *sorted-list-len-lt*)

lemma *sorted-list-psub*:
sorted w $\longrightarrow$
$w \neq [] \longrightarrow$
$(sorted\text{-}list\text{-}of\text{-}set\ (set\ (tl\ w)),\ w) \in (\lambda(x,\ y).\ (sorted\text{-}list\text{-}of\text{-}set\ x,\ sorted\text{-}list\text{-}of\text{-}set\ y))\ \text{`}\ \{(A,\ B).\ A \subset B \wedge finite\ B\}$
proof(*intro strip*, *simp add*:*image-iff*)
assume *sorted-w*: *sorted w* **and** *w-n-nil*: $w \neq []$
let $?a = set\ (tl\ w)$
let $?b = set\ w$

from *sorted-w* **have** *sorted-tl-w*: *sorted* (*tl w*) **and** *dist*: *distinct w* **by** (*induct w*) (*auto simp add*: *sorted-wrt-append*)
with *w-n-nil* **have** *a-psub-b*: $?a \subset ?b$ **by** (*induct w*)*auto*
from *sorted-w sorted-tl-w* **have** $w = sorted\text{-}list\text{-}of\text{-}set\ ?b$ **and** $tl\ w = sorted\text{-}list\text{-}of\text{-}set\ (set\ (tl\ w))$
by (*auto simp add*: *sorted-less-set-eq*)
with *a-psub-b* **show** $\exists\, a\ b.\ a \subset b \wedge finite\ b \wedge sorted\text{-}list\text{-}of\text{-}set\ (set\ (tl\ w)) = sorted\text{-}list\text{-}of\text{-}set\ a \wedge w = sorted\text{-}list\text{-}of\text{-}set\ b$
by *auto*
qed

locale *dom-sl* = *cfg-doms* +
fixes *A* **and** *r* **and** *f* **and** *step* **and** *start* **and** *n*
defines $A \equiv ((rev \circ sorted\text{-}list\text{-}of\text{-}set)\ \text{`}\ (Pow\ (set\ (nodes))))$
defines $r \equiv nodes\text{-}le$
defines $f \equiv nodes\text{-}sup$
defines $n \equiv (size\ nodes)$
defines $step \equiv exec$
defines $start \equiv ([]\ \#\ (replicate\ (length\ (g\text{-}V\ G) - 1)\ (\ (rev[0..<n]))))::state\text{-}dom\ list$

begin

lemma *is-semi*: *semilat*(*A*,*r*,*f*)
by(*insert nodes-semi-is-semilat*) (*auto simp add*:*nodes-semi-def A-def r-def f-def*)

— used by acc_le_listI
lemma *Cons-less-Conss* [*simp*]:
$x\#xs\ [\sqsubset_r]\ y\#ys = (x \sqsubset_r y \wedge xs\ [\sqsubseteq_r]\ ys \vee x = y \wedge xs\ [\sqsubset_r]\ ys)$
apply (*unfold lesssub-def*)
apply *auto*
apply (*unfold lesssub-def lesub-def r-def*)
apply (*simp only*: *nodes-le-refl*)

done

lemma *acc-le-listI* [*intro*!]:
acc r $\Longrightarrow$ *acc* (*Listn.le r*)
apply (*unfold acc-def*)
apply (*subgoal-tac Wellfounded.wf*(*UN n*. {(*ys*,*xs*). *size xs* = *n* ∧ *size ys* = *n* ∧ *xs* <-(*Listn.le r*) *ys*}))
apply (*erule wf-subset*)
apply (*blast intro*: *lesssub-lengthD*)
apply (*rule wf-UN*)
prefer *2*
apply (*rename-tac m n*)
apply (*case-tac m*=*n*)
apply *simp*
apply (*fast intro*!: *equals0I dest*: *not-sym*)
apply (*rename-tac n*)
apply (*induct-tac n*)
apply (*simp add*: *lesssub-def cong*: *conj-cong*)
apply (*rename-tac k*)
apply (*simp add*: *wf-eq-minimal*)
apply (*simp* (*no-asm*) *add*: *length-Suc-conv cong*: *conj-cong*)
apply *clarify*
apply (*rename-tac M m*)
apply (*case-tac* ∃ *x xs*. *size xs* = *k* ∧ *x*#*xs* ∈ *M*)
prefer *2*
apply (*erule thin-rl*)
apply (*erule thin-rl*)
apply *blast*
apply (*erule-tac x* = {*a*. ∃ *xs*. *size xs* = *k* ∧ *a*#*xs*:*M*} **in** *allE*)
apply (*erule impE*)
apply *blast*
apply (*thin-tac* ∃ *x xs*. *P x xs* **for** *P*)
apply *clarify*
apply (*rename-tac maxA xs*)
apply (*erule-tac x* = {*ys*. *size ys* = *size xs* ∧ *maxA*#*ys* ∈ *M*} **in** *allE*)
apply (*erule impE*)
apply *blast*
apply *clarify*
apply (*thin-tac m* ∈ *M*)
apply (*thin-tac maxA*#*xs* ∈ *M*)
apply (*rule bexI*)
prefer *2*
apply *assumption*
apply *clarify*
apply *simp*
apply *blast*
done

lemma *wf-listn*: *wf* {(*y*,*x*). $x \sqsubset_{Listn.le\ r} y$}

by(*insert acc-nodes-le acc-le-listI r-def*) (*simp add*:*acc-def*)

lemma *wf-listn'*: *wf* {(*y*,*x*). *x* [⊏ᵣ] *y*}
by (*rule wf-listn*)

lemma *wf-listn-termination-rel*:
wf ({(*y*,*x*). *x* $\sqsubset_{Listn.le\ r}$ *y*} <*lex*> (λ(*x*, *y*). (*sorted-list-of-set x*, *sorted-list-of-set y*)) ‘ *finite-psubset*)
by (*insert wf-listn wf-sorted-list*) (*fastforce dest*:*wf-lex-prod*)

lemma *inA-is-sorted*: *xs* ∈ *A* ⟹ *sorted* (*rev xs*)
by (*auto simp add*:*A-def sorted-less-sorted-list-of-set*)

lemma *list-nA-lt-refl*: *xs* ∈ *nlists n A* ⟶ *xs* [$\sqsubseteq_r$] *xs*
proof
assume *xs* ∈ *nlists n A*
then have *set xs* ⊆ *A* **by** (*rule nlistsE-set*)
then have ∀ *i*<*length xs*. *xs*!*i* ∈ *A* **by** *auto*
then have ∀ *i*<*length xs*. *sorted* (*rev* (*xs*!*i*)) **by** (*simp add*:*inA-is-sorted*)
then show *xs* [$\sqsubseteq_r$] *xs* **by**(*unfold Listn.le-def lesub-def*)
(*auto simp add*:*list-all2-conv-all-nth Listn.le-def r-def nodes-le-def*)
qed

lemma *nil-inA*: [] ∈ *A*
apply (*unfold A-def*)
apply (*subgoal-tac* {} ∈ *Pow* (*set nodes*))
apply (*subgoal-tac* [] = (λ*x*. *rev* (*sorted-list-of-set x*)) {})
apply (*fastforce intro*:*rev-image-eqI*)
by *auto*

lemma *upt-n-in-pow-nodes*: {*0*..<*n*} ∈ *Pow* (*set nodes*)
by(*auto simp add*:*n-def nodes-def verts-set*)

lemma *rev-all-inA*: *rev* [*0*..<*n*] ∈ *A*
proof(*unfold A-def*,*simp*)
let *?f* = λ*x*. *rev* (*sorted-list-of-set x*)
have *rev* [*0*..<*n*] =*?f* {*0*..<*n*} **by** *auto*
with *upt-n-in-pow-nodes* **show** *rev* [*0*..<*n*] ∈ *?f* ‘ *Pow* (*set nodes*)
by (*fastforce intro*: *image-eqI*)
qed

lemma *len-start-is-n*: *length start* = *n*
by (*insert len-verts-gt0*) (*auto simp add*:*start-def n-def nodes-def dest*:*Suc-pred*)

lemma *len-start-is-len-verts*: *length start* = *length* (*g-V G*)
using *len-verts-gt0* **by** (*simp add*:*start-def*)

lemma *start-len-gt-0*: *length start* > *0*
by (*insert len-verts-gt0*) (*simp add*:*start-def*)

lemma *start-subset-A*: *set start* ⊆ *A*
by(*auto simp add*:*nil-inA rev-all-inA start-def*)

lemma *start-in-A* : *start* ∈ (*nlists n A*)
by (*insert start-subset-A len-start-is-n*)(*fastforce intro*:*nlistsI*)

lemma *sorted-start-nth*: *i* < *n* ⟹ *sorted* (*rev* (*start*!*i*))
apply(*subgoal-tac start*!*i* ∈ *A*)
apply (*fastforce dest*:*inA-is-sorted*)
by (*auto simp add*:*start-subset-A len-start-is-n*)

lemma *start-nth0-empty*: *start*!*0* = []
by (*simp add*:*start-def*)

lemma *start-nth-lt0-all*: ∀ *p*∈{*1*..< *length start*}. *start*!*p* = (*rev* [*0*..<*n*])
by (*auto simp add*:*start-def*)

lemma *in-nodes-lt-n*: *x* ∈ *set* (*g-V G*) ⟹ *x* < *n*
by (*simp add*:*n-def nodes-def verts-set*)

lemma *start-nth0-unstable-auxi*: ¬ [*0*] $\sqsubseteq_r$ (*rev* [*0*..<*n*])
by (*insert len-verts-gt1 verts-ge-Suc0*)
(*auto simp add*:*r-def lesssub-def lesub-def nodes-le-def n-def nodes-def*)

lemma *start-nth0-unstable*: ¬ *stable r step start 0*
proof(*rule notI*,*auto simp add*: *start-nth0-empty stable-def step-def exec-def transf-def*)

assume *ass*: ∀ *x*∈*set* (*sorted-list-of-set* (*succs 0*)). [*0*] $\sqsubseteq_r$ *start* ! *x*
from *succ-of-entry0* **obtain** *s* **where** *s* ∈ *succs 0* **and** *s* ≠ *0* ∧ *s* ∈ *set* (*g-V G*)
using *head-is-vert*
by (*auto simp add*:*succs-def*)
then have *s* ∈ *set* (*sorted-list-of-set* (*succs 0*))
and *start*!*s* = (*rev* [*0*..<*n*]) **using** *fin-succs verts-set len-verts-gt0* **by** (*auto simp add*:*start-def*)
then show *False* **using** *ass start-nth0-unstable-auxi* **by** *auto*
qed

lemma *start-nth-unstable*:
assumes *p* ∈ {*1* ..< *length* (*g-V G*)}
and *succs p* ≠ {}
shows ¬ *stable r step start p*
proof (*rule notI*, *unfold stable-def*)
let *?step-p* = *step p* (*start* ! *p*)
let *?rev-all* = *rev*[*0*..<*length*(*g-V G*)]
assume *sta*: ∀ (*q*, τ)∈*set ?step-p*. τ $\sqsubseteq_r$ *start* ! *q*

from *assms*(*1*) **have** *n-sorted*: ¬ *sorted* (*rev* (*p* # *?rev-all*))
and *p*:*p* ∈ *set* (*g-V G*) **and** *start*!*p* = *?rev-all* **using** *verts-set* **by**

(*auto simp add*:*n-def nodes-def start-def sorted-wrt-append*)
with *sta* **have** *step-p*: $\forall (q, \tau) \in$*set ?step-p. sorted* (*rev* (*p* # *?rev-all*)) $\vee$ (*p* # *?rev-all* = *start*!*q*)
by (*auto simp add*:*step-def exec-def transf-def lesssub-def lesub-def r-def nodes-le-def*)

from *assms*(*2*) *fin-succs p* **obtain** *a b* **where** *a-b*: (*a*, *b*) $\in$ *set ?step-p* **by** (*auto simp add*:*step-def exec-def transf-def*)
with *step-p* **have** *sorted* (*rev* (*p* # *?rev-all*)) $\vee$ (*p* # *?rev-all* = *start*!*a*) **by** *auto*
with *n-sorted* **have** *eq-p-cons*: (*p* # *?rev-all* = *start*!*a*) **by** *auto*

from *p* **have** $\forall (q, \tau) \in$*set ?step-p.* $q < n$ **using** *succ-in-G fin-succs verts-set n-def nodes-def* **by** (*auto simp add*:*step-def exec-def*)
with *a-b* **have** $a < n$ **using** *len-start-is-n* **by** *auto*
then have *sorted* (*rev* (*start*!*a*)) **using** *sorted-start-nth* **by** *auto*
with *eq-p-cons n-sorted* **show** *False* **by** *auto*

qed

lemma *start-unstable-cond*:
assumes *succs p* $\neq$ {}
and $p <$ *length* (*g-V G*)
shows $\neg$ *stable r step start p*
using *assms start-nth0-unstable start-nth-unstable*
by(*cases p* = *0*) *auto*

lemma *unstable-start*: *unstables r step start* = *sorted-list-of-set* ({*p. succs p* $\neq$ {} $\wedge$ $p <$ *length start*})
using *len-start-is-len-verts start-unstable-cond*
by (*subgoal-tac* {*p.* $p <$ *length start* $\wedge$ $\neg$ *stable r step start p*} = {*p. succs p* $\neq$ {} $\wedge$ $p <$ *length start*})
(*auto simp add*: *unstables-def stable-def step-def exec-def*)

end

declare *sorted-list-of-set-insert-remove*[*simp del*]

context *dom-sl*
begin

lemma (**in** *dom-sl*) *decomp-propa*: $\bigwedge$*ss w.*
($\forall (q,t) \in$*set qs.* $q <$ *size ss* $\wedge$ $t \in A$) $\Longrightarrow$
sorted w $\Longrightarrow$
set ss $\subseteq A \Longrightarrow$
propa f qs ss w = (*merges f qs ss*, (*sorted-list-of-set* ({*q.* $\exists t.(q,t) \in$*set qs* $\wedge$ $t \sqcup_f$ (*ss*!*q*) $\neq$ *ss*!*q*} $\cup$ *set w*)))
lemma (**in** *Semilat*)*list-update-le-listI* [*rule-format*]:
set xs $\subseteq A \longrightarrow$ *set ys* $\subseteq A \longrightarrow$ *xs* $[\sqsubseteq_r]$ *ys* $\longrightarrow p <$ *size xs* $\longrightarrow$
$x \sqsubseteq_r$ *ys*!*p* $\longrightarrow x \in A \longrightarrow$
xs[$p := x \sqcup_f$ *xs*!*p*] $[\sqsubseteq_r]$ *ys*

7 Soundness and completeness

```
theory Dom-Kildall-Correct
imports Dom-Kildall-Property
begin

context dom-sl
begin

lemma entry-dominate-dom:
  assumes i ∈ set (g-V G)
      and dominate i 0
    shows dom i 0
  using assms
proof−
  from assms(1) entry0-dominates-all have dominate 0 i by auto
  with assms(2) reachable have i = 0 using reachable-dom-acyclic by (auto simp
add:reachable-def)
  then show ?thesis using dom-refl by auto
qed

lemma path-entry-dom:
  fixes pa i d
  assumes path-entry (g-E G) pa i
      and dom d i
    shows d ∈ set pa ∨ d = i
  using assms
proof(induct rule:path-entry.induct)
  case path-entry0
  then show ?case using zero-dom-zero by auto
next
  case (path-entry-prepend u v l)
  note u-v = path-entry-prepend.hyps(1)
  note ind = path-entry-prepend.hyps(3)
  note d-v = path-entry-prepend.prems
  show ?case
  proof(cases d ≠ v)
    case True note d-n-v = this
    from u-v have v ∈ succs u by (simp add:succs-def)
    with d-v d-n-v have dom d u by (auto intro:adom-succs)
    with ind have d ∈ set l ∨ d = u by auto
    then show ?thesis by auto
  next
    case False
    then show ?thesis by auto
  qed
qed
```

— soundenss

lemma *dom-sound*: *dom i j* ⟹ *dominate i j*
by (*fastforce simp add*: *dominate-def dest*:*path-entry-dom*)

lemma *sdom-sound*: *strict-dom i j* ⟹ *j* ∈ *set* (*g-V G*) ⟹ *strict-dominate i j*
proof −
assume *sdom*: *strict-dom i j* **and** *j* ∈ *set* (*g-V G*)
then have *i-n-j*: *i* ≠ *j* **by** (*rule sdom-asyc*)
from *sdom* **have** *dom i j* **using** *sdom-dom* **by** *auto*
then have *domi*: *dominate i j* **by** (*rule dom-sound*)
with *i-n-j* **show** *?thesis* **by** (*fastforce dest*: *dominate-sdominate*)
qed

— completeness

lemma *dom-complete-auxi*: *i* < *length start* ⟹ (*dom-kildall start*)!*i* = *ss′* ∧ *k* ∉ *set ss′* ⟹ *non-strict-dominate k i*
proof−
assume *i-lt*: *i* < *length start* **and** *dom-kil*: (*dom-kildall start*)!*i* = *ss′*∧ *k* ∉ *set ss′*
then have *dom-iter*: (*fst* (*iter f step start* (*unstables r step start*)))!*i* = *ss′* **and** *k-nin*: *k* ∉ *set ss′*
using *nodes-semi-def r-def f-def step-def dom-kildall-def kildall-def* **by** *auto*
then obtain *s w* **where** *iter*: *iter f step start* (*unstables r step start*) = (*s*, *w*) **by** *fastforce*
with *dom-iter* **have** *s*!*i* = *ss′* **by** *auto*
with *iter-dom-invariant-complete iter k-nin i-lt len-start-is-n*
show *?thesis* **by** *auto*
qed

lemma *notsdom-notsdominate*: ¬ *strict-dom i j* ⟹ *j* < *length start* ⟹ *non-strict-dominate i j*
proof−
assume *i-not-sdom-j*: ¬ *strict-dom i j* **and** *j-lt*: *j* < *length start*
then obtain *res* **where** *j-res*: *dom-kildall start* ! *j* = *res* **by** (*auto simp add*:*strict-dom-def*)
then have *strict-dom i j* = (*i* ∈ *set res*) **by** (*auto simp add*:*strict-dom-def start-def n-def nodes-def*)
with *i-not-sdom-j* **have** *i-nin*: *i* ∉ *set res* **by** *auto*
with *j-res j-lt* **show** *non-strict-dominate i j* **using** *dom-complete-auxi* **by** *fastforce*
qed

lemma *notsdom-notsdominate′*: ¬ *strict-dom i j* ⟹ *j* < *length start* ⟹ ¬ *strict-dominate i j*
using *notsdom-notsdominate nonstrict-eq* **by** *auto*

lemma *dom-complete*: *strict-dominate i j* ⟹ *j* < *size start* ⟹ *strict-dom i j*
by (*insert notsdom-notsdominate′*) (*auto intro*: *contrapos-nn nonstrict-eq*)

end

end

References

[1] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Technical report, Rice University, Houston, Jan. 2006. https://scholarship.rice.edu/handle/1911/96345.

[2] P. Lammich. Operations on sorted lists. 2009. https://www.isa-afp.org/browser_info/current/AFP/Collections/Sorted_List_Operations.html.

[3] T. Nipkow and G. Klein. Operations on sorted lists. 2000. https://www.isa-afp.org/browser_info/current/AFP/Jinja/Kildall.html.