blockDAGs

Jörn

August 30, 2021

Contents

theory *Utils*
imports *Main*
begin

The following functions transform a list L to a relation containing a tuple (a, b) iff $a = b$ or a precedes b in the list L

fun *list-to-rel*:: *'a list* ⇒ *'a rel*
where *list-to-rel* [] = {}
| *list-to-rel* (*x*#*xs*) = {*x*} × (*set* (*x*#*xs*)) ∪ *list-to-rel xs*

lemma *list-to-rel-in* : (*a*,*b*) ∈ (*list-to-rel L*) ⟶ *a* ∈ *set L* ∧ *b* ∈ *set L*
proof(*induct L*, *auto*) **qed**

Show soundness of list-to-rel

lemma *list-to-rel-equal*:
(*a*,*b*) ∈ *list-to-rel L* ⟷ (∃ *k*::*nat*. *hd* (*drop k L*) = *a* ∧ *b* ∈ *set* (*drop k L*))
proof(*safe*)
assume (*a*, *b*) ∈ *list-to-rel L*
then show ∃ *k*. *hd* (*drop k L*) = *a* ∧ *b* ∈ *set* (*drop k L*)
proof(*induct L*)
case *Nil*
then show *?case* **by** *auto*
next

```
    case (Cons a2 L)
    then consider (a, b) ∈ {a2} × set (a2 # L)  | (a,b) ∈  list-to-rel L by auto
    then show ?case unfolding list-to-rel.simps(2)
    proof(cases)
      case 1
      then have a = hd (a2 # L) by auto
      moreover have b ∈ set (a2 # L) using 1 by auto
      ultimately show ?thesis using drop0
        by metis
    next
      case 2
      then obtain k where k-in : hd (drop k (L)) = a ∧ b ∈ set (drop k (L))
        using Cons(1) by auto
      show ?thesis proof
        let ?k = Suc k
        show hd (drop ?k (a2 # L)) = a ∧ b ∈ set (drop ?k (a2 # L))
          unfolding drop-Suc using k-in by auto
      qed
    qed
  qed
next
  fix k
  assume b ∈ set (drop k L)
    and a = hd (drop k L)
  then show (hd (drop k L), b) ∈ list-to-rel L
  proof(induct L arbitrary: k)
    case Nil
    then show ?case by auto
  next
    case (Cons a L)
    consider (zero) k = 0 | (more) k > 0 by auto
    then show ?case
    proof(cases)
      case zero
      then show ?thesis using Cons drop-0 by auto
    next
      case more
      then obtain k2 where k2-in:  k = Suc k2
        using gr0-implies-Suc by auto
       show ?thesis using Cons unfolding k2-in drop-Suc list-to-rel.simps(2) by
auto
    qed
  qed
qed

lemma list-to-rel-append:
  assumes a ∈ set L
  shows (a,b) ∈ list-to-rel (L @ [b])
  using assms
```

proof(*induct L*, *simp*, *auto*) **qed**

For every distinct L, list-to-rel L return a linear order on set L

lemma *list-order-linear*:
assumes *distinct L*
shows *linear-order-on* (*set L*) (*list-to-rel L*)
unfolding *linear-order-on-def total-on-def partial-order-on-def preorder-on-def refl-on-def*
trans-def antisym-def
proof(*safe*)
fix *a b*
assume (*a*, *b*) ∈ *list-to-rel L*
then show *a* ∈ *set L*
proof(*induct L*, *auto*) **qed**
next
fix *a b*
assume (*a*, *b*) ∈ *list-to-rel L*
then show *b* ∈ *set L*
proof(*induct L*, *auto*) **qed**
next
fix *x*
assume *x* ∈ *set L*
then show (*x*, *x*) ∈ *list-to-rel L*
proof(*induct L*, *auto*) **qed**
next
fix *x y z*
assume *as1*: (*x*,*y*) ∈ *list-to-rel L*
and *as2*: (*y*, *z*) ∈ *list-to-rel L*
then show (*x*, *z*) ∈ *list-to-rel L*
using *assms*
proof(*induct L*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a L*)
then consider (*nor*) (*x*, *y*) ∈ {*a*} × *set* (*a* # *L*) ∧ (*y*, *z*) ∈ {*a*} × *set* (*a* # *L*)
| (*xy*) (*x*,*y*) ∈ *list-to-rel L* ∧ (*y*, *z*) ∈ {*a*} × *set* (*a* # *L*)
| (*yz*) (*y*,*z*) ∈ *list-to-rel L* ∧ (*x*, *y*) ∈ {*a*} × *set* (*a* # *L*)
| (*both*) (*y*,*z*) ∈ *list-to-rel L* ∧ (*x*,*y*) ∈ *list-to-rel L* **by** *auto*
then show *?case* **proof**(*cases*)
case *nor*
then show *?thesis* **by** *auto*
next
case *xy*
then have *y* ∈ *set L* **using** *list-to-rel-in* **by** *metis*
also have *y* = *a* **using** *xy* **by** *auto*
ultimately have ¬ *distinct* (*a* # *L*)
by *simp*
then show *?thesis* **using** *Cons* **by** *auto*

```
    next
      case yz
      then show ?thesis using list-to-rel.simps(2)
        by (metis Cons.prems(2) SigmaD1 SigmaI UnI1 list-to-rel-in)
    next
      case both
      then show ?thesis unfolding list-to-rel.simps(2) using Cons by auto
    qed
  qed
next
  fix x y
  assume (x, y) ∈ list-to-rel L
    and (y, x) ∈ list-to-rel L
  then show x = y
    using assms
  proof(induct L, simp)
    case (Cons a L)
   then consider (nor) (x, y) ∈ {a} × set (a # L) ∧ (y, x) ∈ {a} × set (a # L)
      | (xy) (x,y) ∈ list-to-rel L ∧ (y, x) ∈ {a} × set (a # L)
      | (yz) (y,x) ∈ list-to-rel L ∧ (x, y) ∈ {a} × set (a # L)
      | (both) (y,x) ∈ list-to-rel L ∧ (x,y) ∈ list-to-rel L by auto
    then show ?case unfolding list-to-rel.simps
    proof(cases)
      case nor
      then show ?thesis by auto
    next
      case xy
      then show ?thesis
       by (metis Cons.prems(3) SigmaD1 distinct.simps(2) list-to-rel-in singletonD)

    next
      case yz
      then show ?thesis
       by (metis Cons.prems(3) SigmaD1 distinct.simps(2) list-to-rel-in singletonD)

    next
      case both
      then show ?thesis using Cons by auto
    qed
  qed
next
  fix x y
  assume x ∈ set L
    and y ∈ set L
    and x ≠ y
    and (y, x) ∉ list-to-rel L
  then show (x, y) ∈ list-to-rel L
  proof(induct L, auto) qed
qed
```

lemma *list-to-rel-mono*:
assumes (*a*,*b*) ∈ *list-to-rel* (*L*)
shows (*a*,*b*) ∈ *list-to-rel* (*L* @ *L2*)
using *assms*
proof(*induct L2 arbitrary*: *L*, *simp*)
case (*Cons a L2*)
then show *?case*
proof(*induct L*, *auto*)
qed
qed

lemma *list-to-rel-mono2*:
assumes (*a*,*b*) ∈ *list-to-rel* (*L2*)
shows (*a*,*b*) ∈ *list-to-rel* (*L* @ *L2*)
using *assms*
proof(*induct L2 arbitrary*: *L*, *simp*)
case (*Cons a L2*)
then show *?case*
proof(*induct L*, *auto*)
qed
qed

lemma *map-snd-map*: ⋀*L*. (*map snd* (*map* (λ*i*. (*P i* , *i*)) *L*)) = *L*
proof −
fix *L*
show *map snd* (*map* (λ*i*. (*P i*, *i*)) *L*) = *L*
proof(*induct L*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a L*)
then show *?case* **by** *auto*
qed
qed
end

theory *DigraphUtils*
imports *Main Graph-Theory.Graph-Theory*
begin

1 Digraph Utilities

lemma *graph-equality*:

assumes *digraph G* ∧ *digraph C*
assumes *verts G* = *verts C* ∧ *arcs G* = *arcs C* ∧ *head G* = *head C* ∧ *tail G* = *tail C*
shows *G* = *C*
by (*simp add*: *assms*(*2*))

lemma (**in** *digraph*) *del-vert-not-in-graph*:
assumes *b* ∉ *verts G*
shows (*pre-digraph.del-vert G b*) = *G*
proof −
have *v*: *verts* (*pre-digraph.del-vert G b*) = *verts G*
using *assms*(*1*)
by (*simp add*: *pre-digraph.verts-del-vert*)
have ∀ *e* ∈ *arcs G*. *tail G e* ≠ *b* ∧ *head G e* ≠ *b* **using** *digraph-axioms assms digraph.axioms*(*2*) *loopfree-digraph.axioms*(*1*)
by *auto*
then have *arcs G* ⊆ *arcs* (*pre-digraph.del-vert G b*)
using *assms*
by (*simp add*: *pre-digraph.arcs-del-vert subsetI*)
then have *e*: *arcs G* = *arcs* (*pre-digraph.del-vert G b*)
by (*simp add*: *pre-digraph.arcs-del-vert subset-antisym*)
then show *?thesis* **using** *v* **by** (*simp add*: *pre-digraph.del-vert-simps*)
qed

lemma *del-arc-subgraph*:
assumes *subgraph H G*
assumes *digraph G* ∧ *digraph H*
shows *subgraph* (*pre-digraph.del-arc H e2*) (*pre-digraph.del-arc G e2*)
using *subgraph-def pre-digraph.del-arc-simps Diff-iff*
proof −
have *f1*: ∀ *p pa*. *subgraph p pa* = ((*verts p*::′*a set*) ⊆ *verts pa* ∧ (*arcs p*::′*b set*) ⊆ *arcs pa* ∧
wf-digraph pa ∧ *wf-digraph p* ∧ *compatible pa p*)
using *subgraph-def* **by** *blast*
have *arcs H* − {*e2*} ⊆ *arcs G* − {*e2*} **using** *assms*(*1*)
by *auto*
then show *?thesis*
unfolding *subgraph-def*
using *f1 assms*(*1*) **by** (*simp add*: *compatible-def pre-digraph.del-arc-simps wf-digraph.wf-digraph-del-arc*)
qed

lemma *graph-nat-induct*[*consumes 0*, *case-names base step*]:
assumes

cases: ⋀*V*. (*digraph V* ⟹ *card* (*verts V*) = *0* ⟹ *P V*)
⋀*W c*. (⋀*V*. (*digraph V* ⟹ *card* (*verts V*) = *c* ⟹ *P V*))

$\Longrightarrow$ (*digraph W* $\Longrightarrow$ *card* (*verts W*) = (*Suc c*) $\Longrightarrow$ *P W*)
shows $\bigwedge Z$. *digraph Z* $\Longrightarrow$ *P Z*
proof −
fix *Z*:: ('*a*,'*b*) *pre-digraph*
assume *major*: *digraph Z*
then show *P Z*
proof (*induction card* (*verts Z*) *arbitrary*: *Z*)
case *0*
then show *?case*
by (*simp add*: *local.cases*(*1*) *major*)
next
case *su*: (*Suc x*)
assume ($\bigwedge Z$. *x* = *card* (*verts Z*) $\Longrightarrow$ *digraph Z* $\Longrightarrow$ *P Z*)
show *?case*
by (*metis local.cases*(*2*) *su.hyps*(*1*) *su.hyps*(*2*) *su.prems*)
qed
qed
end

theory *DAGs*
imports *Main Graph-Theory.Graph-Theory*
begin

2 DAG

locale *DAG* = *digraph* +
assumes *cycle-free*: ¬(*v* $\rightarrow^+_G$ *v*)

sublocale *DAG* ⊆ *wf-digraph* **using** *DAG-def digraph-def nomulti-digraph-def DAG-axioms* **by** *auto*

2.1 Functions and Definitions

fun *direct-past*:: ('*a*,'*b*) *pre-digraph* ⇒ '*a* ⇒ '*a set*
where *direct-past G a* = {*b* ∈ *verts G*. (*a*,*b*) ∈ *arcs-ends G*}

fun *future-nodes*:: ('*a*,'*b*) *pre-digraph* ⇒ '*a* ⇒ '*a set*
where *future-nodes G a* = {*b* ∈ *verts G*. *b* $\rightarrow^+_G$ *a*}

fun *past-nodes*:: ('*a*,'*b*) *pre-digraph* ⇒'*a* ⇒ '*a set*
where *past-nodes G a* = {*b* ∈ *verts G*. *a* $\rightarrow^+_G$ *b*}

fun *past-nodes-refl* :: ('*a*,'*b*) *pre-digraph* ⇒ '*a* ⇒ '*a set*
where *past-nodes-refl G a* = {*b* ∈ *verts G*. *a* $\rightarrow^*_G$ *b*}

fun *anticone*:: ('*a*,'*b*) *pre-digraph* ⇒ '*a* ⇒ '*a set*
where *anticone G a* = {*b* ∈ *verts G*. ¬(*a* $\rightarrow^+_G$ *b* ∨ *b* $\rightarrow^+_G$ *a* ∨ *a* = *b*)}

fun *reduce-past*:: ('*a*,'*b*) *pre-digraph* ⇒ '*a* ⇒ ('*a*,'*b*) *pre-digraph*
where
reduce-past G a = *induce-subgraph G* (*past-nodes G a*)

fun *reduce-past-refl*:: ('*a*,'*b*) *pre-digraph* ⇒ '*a* ⇒ ('*a*,'*b*) *pre-digraph*
where
reduce-past-refl G a = *induce-subgraph G* (*past-nodes-refl G a*)

fun *is-tip*:: ('*a*,'*b*) *pre-digraph* ⇒ '*a* ⇒ *bool*
where *is-tip G a* = ((*a* ∈ *verts G*) ∧ (∀ *x* ∈ *verts G*. ¬ *x* →⁺ *G a*))

definition *tips*:: ('*a*,'*b*) *pre-digraph* ⇒ '*a set*
where *tips G* = {*v* ∈ *verts G*. *is-tip G v*}

fun *kCluster*:: ('*a*,'*b*) *pre-digraph* ⇒ *nat* ⇒ '*a set* ⇒ *bool*
where *kCluster G k C* = (*if* (*C* ⊆ (*verts G*))
then (∀ *a* ∈ *C*. *card* ((*anticone G a*) ∩ *C*) ≤ *k*) *else False*)

2.2 Lemmas

lemma (**in** *DAG*) *unidirectional*:
u →⁺ *G v* ⟶ ¬(*v* →* *G u*)
using *cycle-free reachable1-reachable-trans* **by** *auto*

2.2.1 Tips

lemma (**in** *wf-digraph*) *tips-not-referenced*:
assumes *is-tip G t*
shows ∀ *x*. ¬ *x* →⁺ *t*
using *is-tip.simps assms reachable1-in-verts*(*1*)
by *metis*

lemma (**in** *DAG*) *del-tips-dag*:
assumes *is-tip G t*
shows *DAG* (*del-vert t*)
unfolding *DAG-def DAG-axioms-def*
proof *safe*
show *digraph* (*del-vert t*) **using** *del-vert-simps DAG-axioms*
digraph-def
using *digraph-subgraph subgraph-del-vert*
by *auto*
next
fix *v*
assume *v* →⁺ *del-vert t v*
then have *v* →⁺ *v* **using** *subgraph-del-vert*
by (*meson arcs-ends-mono trancl-mono*)
then show *False*
by (*simp add*: *cycle-free*)
qed

lemma (**in** *digraph*) *tips-finite*:
shows *finite* (*tips G*)
using *tips-def fin-digraph.finite-verts digraph.axioms*(*1*) *digraph-axioms Collect-mono is-tip.simps*
by (*simp add*: *tips-def*)

lemma (**in** *digraph*) *tips-in-verts*:
shows *tips G* ⊆ *verts G* **unfolding** *tips-def*
using *Collect-subset* **by** *auto*

lemma *tips-tips*:
assumes *x* ∈ *tips G*
shows *is-tip G x* **using** *tips-def CollectD assms*(*1*) **by** *metis*

2.2.2 Anticone

lemma (**in** *DAG*) *tips-anticone*:
assumes *a* ∈ *tips G*
and *b* ∈ *tips G*
and *a* ≠ *b*
shows *a* ∈ *anticone G b*
proof(*rule ccontr*)
assume *a* ∉ *anticone G b*
then have *k*: (*a* →⁺ *b* ∨ *b* →⁺ *a* ∨ *a* = *b*) **using** *anticone.simps assms tips-def*
by *fastforce*
then have ¬ (∀ *x*∈*verts G*. *x* →⁺ *a*) ∨ ¬ (∀ *x*∈*verts G*. *x* →⁺ *b*) **using** *reachable1-in-verts*
assms(*3*) *cycle-free*
by (*metis*)
then have ¬ *is-tip G a* ∨ ¬ *is-tip G b* **using** *assms*(*3*) *is-tip.simps k*
by (*metis*)
then have ¬ *a* ∈ *tips G* ∨ ¬ *b* ∈ *tips G* **using** *tips-def CollectD* **by** *metis*
then show *False* **using** *assms* **by** *auto*
qed

lemma (**in** *DAG*) *anticone-in-verts*:
shows *anticone G a* ⊆ *verts G* **using** *anticone.simps* **by** *auto*

lemma (**in** *DAG*) *anticon-finite*:
shows *finite* (*anticone G a*) **using** *anticone-in-verts* **by** *auto*

lemma (**in** *DAG*) *anticon-not-refl*:
shows *a* ∉ (*anticone G a*) **by** *auto*

2.2.3 Future Nodes

lemma (**in** *DAG*) *future-nodes-not-refl*:
assumes *a* ∈ *verts G*
shows *a* ∉ *future-nodes G a*

using *cycle-free future-nodes.simps reachable-def* **by** *auto*

2.2.4 Past Nodes

lemma (**in** *DAG*) *past-nodes-not-refl*:
assumes $a \in verts\ G$
shows $a \notin past\text{-}nodes\ G\ a$
using *cycle-free past-nodes.simps reachable-def* **by** *auto*

lemma (**in** *DAG*) *past-nodes-verts*:
shows $past\text{-}nodes\ G\ a \subseteq verts\ G$
using *past-nodes.simps reachable1-in-verts* **by** *auto*

lemma (**in** *DAG*) *past-nodes-refl-ex*:
assumes $a \in verts\ G$
shows $a \in past\text{-}nodes\text{-}refl\ G\ a$
using *past-nodes-refl.simps reachable-refl assms*
by *simp*

lemma (**in** *DAG*) *past-nodes-refl-verts*:
shows $past\text{-}nodes\text{-}refl\ G\ a \subseteq verts\ G$
using *past-nodes.simps reachable-in-verts* **by** *auto*

lemma (**in** *DAG*) *finite-past*: *finite* (*past-nodes G a*)
by (*metis finite-verts rev-finite-subset past-nodes-verts*)

lemma (**in** *DAG*) *future-nodes-verts*:
shows $future\text{-}nodes\ G\ a \subseteq verts\ G$
using *future-nodes.simps reachable1-in-verts* **by** *auto*

lemma (**in** *DAG*) *finite-future*: *finite* (*future-nodes G a*)
by (*metis finite-verts rev-finite-subset future-nodes-verts*)

lemma (**in** *DAG*) *past-future-dis*[*simp*]: $past\text{-}nodes\ G\ a \cap future\text{-}nodes\ G\ a = \{\}$
proof (*rule ccontr*)
assume $\neg\ past\text{-}nodes\ G\ a \cap future\text{-}nodes\ G\ a = \{\}$
then show *False*
using *past-nodes.simps future-nodes.simps unidirectional reachable1-reachable*
by *auto*
qed

2.2.5 Reduce Past

lemma (**in** *DAG*) *reduce-past-arcs*:
shows $arcs\ (reduce\text{-}past\ G\ a) \subseteq arcs\ G$
using *induce-subgraph-arcs past-nodes.simps* **by** *auto*

lemma (**in** *DAG*) *reduce-past-arcs2*:
$e \in arcs\ (reduce\text{-}past\ G\ a) \Longrightarrow e \in arcs\ G$
using *reduce-past-arcs* **by** *auto*

lemma (**in** *DAG*) *reduce-past-induced-subgraph*:
shows *induced-subgraph* (*reduce-past G a*) *G*
using *induced-induce past-nodes-verts* **by** *auto*

lemma (**in** *DAG*) *reduce-past-path*:
assumes $u \rightarrow^+_{reduce\text{-}past\ G\ a}\ v$
shows $u \rightarrow^+_G v$
using *assms*
proof *induct*
case *base* **then show** *?case*
using *dominates-induce-subgraphD r-into-trancl′ reduce-past.simps*
by *metis*
next case (*step u v*) **show** *?case*
using *dominates-induce-subgraphD reachable1-reachable-trans reachable-adjI*
reduce-past.simps step.hyps(*2*) *step.hyps*(*3*) **by** *metis*

qed

lemma (**in** *DAG*) *reduce-past-path2*:
assumes $u \rightarrow^+_G v$
and $u \in$ *past-nodes G a*
and $v \in$ *past-nodes G a*
shows $u \rightarrow^+_{reduce\text{-}past\ G\ a}\ v$
using *assms*
proof(*induct u v*)
case (*r-into-trancl u v*)
then obtain *e* **where** *e-in*: *arc e* (*u*,*v*) **using** *arc-def DAG-axioms wf-digraph-def*
by *auto*
then have *e-in2*: *e* ∈ *arcs* (*reduce-past G a*) **unfolding** *reduce-past.simps induce-subgraph-arcs*
using *arcE r-into-trancl.prems*(*1*) *r-into-trancl.prems*(*2*) **by** *blast*
then have *arc-to-ends* (*reduce-past G a*) *e* = (*u*,*v*) **unfolding** *reduce-past.simps*
using *e-in*
arcE arc-to-ends-def induce-subgraph-head induce-subgraph-tail
by *metis*

then have $u \rightarrow_{reduce\text{-}past\ G\ a}\ v$ **using** *e-in2 wf-digraph.dominatesI DAG-axioms*
by (*metis reduce-past.simps wellformed-induce-subgraph*)
then show *?case* **by** *auto*
next
case (*trancl-into-trancl a2 b c*)
then have *b-in*: *b* ∈ *past-nodes G a* **unfolding** *past-nodes.simps*
by (*metis* (*mono-tags*, *lifting*) *adj-in-verts*(*1*) *mem-Collect-eq*
reachable1-reachable reachable1-reachable-trans)
then have *a2-re-b*: $a2 \rightarrow^+_{reduce\text{-}past\ G\ a}\ b$ **using** *trancl-into-trancl* **by** *auto*
then obtain *e* **where** *e-in*: *arc e* (*b*,*c*) **using** *trancl-into-trancl*
arc-def DAG-axioms wf-digraph-def **by** *auto*
then have *e-in2*: *e* ∈ *arcs* (*reduce-past G a*) **unfolding** *reduce-past.simps in-*

duce-subgraph-arcs
using *arcE trancl-into-trancl*
b-in **by** *blast*
then have *arc-to-ends* (*reduce-past G a*) *e* = (*b*,*c*) **unfolding** *reduce-past.simps*
using *e-in*
arcE arc-to-ends-def induce-subgraph-head induce-subgraph-tail
by *metis*
then have $b \rightarrow_{reduce\text{-}past\ G\ a}\ c$ **using** *e-in2 wf-digraph.dominatesI DAG-axioms*
by (*metis reduce-past.simps wellformed-induce-subgraph*)
then show *?case* **using** *a2-re-b*
by (*metis trancl.trancl-into-trancl*)
qed

lemma (**in** *DAG*) *reduce-past-pathr*:
assumes $u \rightarrow^*_{reduce\text{-}past\ G\ a}\ v$
shows $u \rightarrow^*_G v$
by (*meson assms induced-subgraph-altdef reachable-mono reduce-past-induced-subgraph*)

2.2.6 Reduce Past Reflexiv

lemma (**in** *DAG*) *reduce-past-refl-induced-subgraph*:
shows *induced-subgraph* (*reduce-past-refl G a*) *G*
using *induced-induce past-nodes-refl-verts* **by** *auto*

lemma (**in** *DAG*) *reduce-past-refl-arcs2*:
$e \in$ *arcs* (*reduce-past-refl G a*) $\Longrightarrow e \in$ *arcs G*
using *reduce-past-arcs* **by** *auto*

lemma (**in** *DAG*) *reduce-past-refl-digraph*:
assumes $a \in$ *verts G*
shows *digraph* (*reduce-past-refl G a*)
using *digraphI-induced reduce-past-refl-induced-subgraph reachable-mono* **by** *simp*

2.2.7 Reachability cases

lemma (**in** *DAG*) *reachable1-cases*:
obtains (*nR*) $\neg\ a \rightarrow^+ b \wedge \neg\ b \rightarrow^+ a \wedge a \neq b$
| (*one*) $a \rightarrow^+ b$
| (*two*) $b \rightarrow^+ a$
| (*eq*) $a = b$
using *reachable-neq-reachable1 DAG-axioms*
by *metis*

lemma (**in** *DAG*) *verts-comp*:
assumes $x \in$ *tips G*
shows *verts G* = $\{x\} \cup$ (*anticone G x*) $\cup$ (*verts* (*reduce-past G x*))
proof
show *verts G* $\subseteq \{x\} \cup$ *anticone G x* $\cup$ *verts* (*reduce-past G x*)

```
  proof(rule subsetI)
    fix xa
    assume in-V: xa ∈ verts G
    then show xa ∈ {x} ∪ anticone G x ∪ verts (reduce-past G x)
    proof( cases x xa rule: reachable1-cases)
      case nR
      then show ?thesis using anticone.simps in-V by auto
    next
      case one
    then show ?thesis using reduce-past.simps induce-subgraph-verts past-nodes.simps
in-V
        by auto
    next
      case two
      have is-tip G x using tips-tips assms(1) by simp
      then have False using  tips-not-referenced two by auto
      then show ?thesis by simp
    next
      case eq
      then show ?thesis by auto
    qed
  qed
next
   show {x} ∪ anticone G x ∪ verts (reduce-past G x) ⊆ verts G using di-
graph.tips-in-verts
     digraph-axioms anticone-in-verts reduce-past-induced-subgraph induced-subgraph-def
       subgraph-def assms by auto
qed


lemma (in DAG) verts-comp2:
  assumes x ∈ tips G
    and a ∈ verts G
  obtains  a = x
  | a ∈ anticone G x
  | a ∈ past-nodes G x
  using assms
proof(cases a x rule:reachable1-cases)
  case one
  then show ?thesis
    by (metis assms(1) tips-not-referenced tips-tips)
next
  case two
 then show ?thesis using past-nodes.simps wf-digraph.reachable1-in-verts(2) wf-digraph-axioms
      mem-Collect-eq that(3)
    by (metis (no-types, lifting))
next
  case nR
  then show ?thesis using that(2) anticone.simps assms by auto
```

qed

lemma (**in** *DAG*) *verts-comp-dis*:
shows $\{x\} \cap (anticone\ G\ x) = \{\}$
and $\{x\} \cap (verts\ (reduce\text{-}past\ G\ x)) = \{\}$
and $anticone\ G\ x \cap (verts\ (reduce\text{-}past\ G\ x)) = \{\}$
proof(*simp-all*, *simp add*: *cycle-free*, *safe*) **qed**

lemma (**in** *DAG*) *verts-size-comp*:
assumes $x \in tips\ G$
shows $card\ (verts\ G) = 1 + card\ (anticone\ G\ x) + card\ (verts\ (reduce\text{-}past\ G\ x))$
proof −
have *f1*: *finite* $(verts\ G)$ **using** *finite-verts* **by** *simp*
have *f2*: *finite* $\{x\}$ **by** *auto*
have *f3*: *finite* $(anticone\ G\ x)$ **using** *anticone.simps* **by** *auto*
have *f4*: *finite* $(verts\ (reduce\text{-}past\ G\ x))$ **by** *auto*
have *c1*: $card\ \{x\} + card\ (anticone\ G\ x) = card\ (\{x\} \cup (anticone\ G\ x))$ **using** *card-Un-disjoint*
verts-comp-dis **by** *auto*
have $(\{x\} \cup (anticone\ G\ x)) \cap verts\ (reduce\text{-}past\ G\ x) = \{\}$ **using** *verts-comp-dis* **by** *auto*
then have $card\ (\{x\} \cup (anticone\ G\ x) \cup verts\ (reduce\text{-}past\ G\ x))$
$= card\ \{x\} + card\ (anticone\ G\ x) + card\ (verts\ (reduce\text{-}past\ G\ x))$
using *card-Un-disjoint*
by (*metis c1 f2 f3 f4 finite-UnI*)
moreover have $card\ (verts\ G) = card\ (\{x\} \cup (anticone\ G\ x) \cup verts\ (reduce\text{-}past\ G\ x))$
using *assms verts-comp* **by** *auto*
moreover have $card\ \{x\} = 1$ **by** *simp*
ultimately show *?thesis* **using** *assms verts-comp*
by *presburger*
qed

end
theory *TopSort*
imports *DAGs Utils*
begin

Function to sort a list L under a graph G such if a references b, b precedes a in the list

fun *top-insert*:: (*'a*::*linorder*,*'b*) *pre-digraph* ⇒*'a list* ⇒ *'a* ⇒ *'a list*
where *top-insert G* [] *a* = [*a*]
| *top-insert G* (*b* # *L*) *a* = (*if* (*b* $\rightarrow^{+}_{G}$ *a*) *then* (*a* # (*b* # *L*)) *else* (*b* # *top-insert G L a*))

fun *top-sort*:: (*'a*::*linorder*,*'b*) *pre-digraph* ⇒ *'a list* ⇒ *'a list*
where *top-sort G* []= []

| *top-sort G (a # L) = top-insert G (top-sort G L) a*

2.2.8 Soundness of the topological sort algorithm

lemma *top-insert-set*: *set (top-insert G L a) = set L ∪ {a}*
proof(*induct L*, *simp-all*, *auto*) **qed**

lemma *top-sort-con*: *set (top-sort G L) = set L*
proof(*induct L*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a L*)
then show *?case* **using** *top-sort.simps*(*2*) *top-insert-set insert-is-Un list.simps*(*15*) *sup-commute*
by (*metis*)
qed

lemma *top-insert-len*: *length (top-insert G L a) = Suc (length L)*
proof(*induct L*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a L*)
then show *?case* **using** *top-insert.simps*(*2*) **by** *auto*
qed

lemma *top-sort-len*: *length (top-sort G L) = length L*
proof(*induct L*, *simp*)
case (*Cons a L*)
then have *length (a#L) = Suc (length L)* **by** *auto*
then show *?case* **using**
top-insert-len top-sort.simps(*2*) *Cons*
by (*simp add*: *top-insert-len*)
qed

lemma *top-insert-mono*:
assumes (*y*, *x*) ∈ *list-to-rel ls*
shows (*y*, *x*) ∈ *list-to-rel (top-insert G ls l)*
using *assms*
proof(*induct ls*, *simp*)
case (*Cons a ls*)
consider (*rec*) *a* $\rightarrow^+{}_G$ *l* | (*nrec*) ¬ *a* $\rightarrow^+{}_G$ *l* **by** *auto*
then show *?case*
proof(*cases*)
case *rec*
then have *sinse*: *(top-insert G (a # ls) l) = l # a # ls*
unfolding *top-insert.simps* **by** *simp*

```
    show ?thesis unfolding sinse list-to-rel.simps  using Cons
      by auto
  next
    case nrec
    then have sinse: (top-insert G (a # ls) l)  = a # top-insert G ls l
      unfolding top-insert.simps by simp
    consider (ya) y = a | (yan) (y, x) ∈ list-to-rel ls using Cons by auto
    then show ?thesis proof(cases)
      case ya
      then show ?thesis unfolding sinse list-to-rel.simps
       by (metis Cons.prems SigmaI UnI1 top-insert-set insertCI list-to-rel-in sinse)

    next
      case yan
      then show ?thesis using Cons unfolding sinse list-to-rel.simps by auto
    qed
  qed
qed

lemma top-sort-mono:
  assumes (y, x) ∈ list-to-rel (top-sort G ls)
  shows (y, x) ∈ list-to-rel (top-sort G (l # ls))
  using assms
  by (simp add: top-insert-mono)



fun (in DAG) top-sorted :: 'a list ⇒ bool where
  top-sorted [] = True |
  top-sorted (x # ys) = ((∀ y ∈ set ys. ¬ x →⁺_G y) ∧ top-sorted ys)

lemma (in DAG) top-sorted-sub:
  assumes S = drop k L
    and top-sorted L
  shows top-sorted S
  using assms
proof(induct k arbitrary: L S)
  case 0
  then show ?case by auto
next
  case (Suc k)
  then show ?case unfolding drop-Suc using top-sorted.simps
    by (metis Suc.prems(1) drop-Nil list.sel(3) top-sorted.elims(2))
qed


lemma top-insert-part-ord:
  assumes DAG G
    and DAG.top-sorted G L
```

```
  shows DAG.top-sorted G (top-insert G L a)
  using assms
proof(induct L)
  case Nil
  then show ?case
    by (simp add: DAG.top-sorted.simps)
next
  case (Cons b list)
  consider (re) b →⁺_G a | (nre) ¬ b →⁺_G a  by auto
  then show ?case proof(cases)
    case re
    have (∀ y∈set (b # list). ¬ a →⁺_G y )
    proof(rule ccontr)
      assume ¬ (∀ y∈set (b # list). ¬ a →⁺_G y)
      then obtain wit where wit-in: wit ∈ set  (b # list) ∧ a →⁺_G wit by auto
      then have b →⁺_G wit using re
        by auto
      then have ¬ DAG.top-sorted G  (b # list)
        using wit-in using DAG.top-sorted.simps(2) Cons(2)
        by (metis DAG.cycle-free set-ConsD)
      then show False using Cons by auto
    qed
    then show ?thesis using assms(1) DAG.top-sorted.simps Cons
      by (simp add: DAG.top-sorted.simps(2) re)
  next
    case nre
    have DAG.top-sorted G list using Cons(2,3)
      by (metis DAG.top-sorted.simps(2))
    then have DAG.top-sorted G (top-insert G list a)
      using  Cons(1,2) by auto
   moreover have (∀ y∈set (top-insert G list a). ¬ b →⁺_G y ) using top-insert-set

       Cons DAG.top-sorted.simps(2) nre
      by (metis Un-iff empty-iff empty-set list.simps(15) set-ConsD)
    ultimately show ?thesis using Cons(2)
      by (simp add: DAG.top-sorted.simps(2) nre)
  qed
qed


lemma top-sort-sorted:
  assumes DAG G
  shows DAG.top-sorted G (top-sort G L)
  using assms
proof(induct L)
  case Nil
  then show ?case
    by (simp add: DAG.top-sorted.simps(1))
  case (Cons a L)
```

then show *?case* **unfolding** *top-sort.simps* **using** *top-insert-part-ord* **by** *auto*
qed

lemma *top-sorted-rel*:
assumes *DAG G*
and $y \rightarrow^+_G x$
and $x \in$ *set L*
and $y \in$ *set L*
and *DAG.top-sorted G L*
shows $(x,y) \in$ *list-to-rel L*
using *assms*
proof(*induct L*, *simp*)
have *une*:$x \neq y$ **using** *assms*
by (*metis DAG.cycle-free*)
case (*Cons a L*)
then consider $x = a \wedge y \in$ *set* $(a \# L) \mid y = a \wedge x \in$ *set L* $\mid x \in$ *set L* $\wedge\ y \in$ *set L*
using *une* **by** *auto*
then show *?case* **proof**(*cases*)
case *1*
then show *?thesis* **unfolding** *list-to-rel.simps* **by** *auto*
next
case *2*
then have $\neg$ *DAG.top-sorted G* $(a \# L)$
using *assms DAG.top-sorted.simps*(*2*)
by *fastforce*
then show *?thesis* **using** *Cons* **by** *auto*
next
case *3*
then show *?thesis* **unfolding** *list-to-rel.simps* **using** *Cons DAG.top-sorted.simps*(*2*) *Un-iff*
by *metis*
qed
qed

lemma *top-sort-rel*:
assumes *DAG G*
and $y \rightarrow^+_G x$
and $x \in$ *set L*
and $y \in$ *set L*
shows $(x,y) \in$ *list-to-rel* (*top-sort G L*)
using *assms top-sort-sorted top-sorted-rel top-sort-con*
by *metis*

end

theory *blockDAG*
imports *DAGs DigraphUtils*

begin

3 blockDAGs

locale *blockDAG* = *DAG* +
  assumes *genesis*: $\exists\, p \in \textit{verts}\ G.\ \forall\, r.\ r \in \textit{verts}\ G \longrightarrow (r \rightarrow^{+}{}_{G}\ p \vee r = p)$
    and *only-new*: $\forall\, e.\ (u \rightarrow^{+}{}_{(\textit{del-arc}\ e)}\ v) \longrightarrow \neg\ \textit{arc}\ e\ (u,v)$
begin

lemma *bD*: *blockDAG G* **using** *blockDAG-axioms* **by** *simp*

end

3.1 Functions and Definitions

fun (**in** *blockDAG*) *is-genesis-node* :: *'a* ⇒ *bool* **where**
  is-genesis-node v = $((v \in \textit{verts}\ G) \wedge (\textit{ALL}\ x.\ (x \in \textit{verts}\ G) \longrightarrow\ x \rightarrow^{*}{}_{G}\ v))$

definition (**in** *blockDAG*) *genesis-node*:: *'a*
  where *genesis-node* = (*THE x. is-genesis-node x*)

3.2 Lemmas

lemma *subs*:
  assumes *blockDAG G*
  shows *DAG G* ∧ *digraph G* ∧ *fin-digraph G* ∧ *wf-digraph G*
  using *assms blockDAG-def DAG-def digraph-def fin-digraph-def* **by** *blast*

3.2.1 Genesis

lemma (**in** *blockDAG*) *genesisAlt* :
  $(\textit{is-genesis-node}\ a) \longleftrightarrow ((a \in \textit{verts}\ G) \wedge (\forall\, r.\ \ (r \in \textit{verts}\ G) \longrightarrow r \rightarrow^{*}\ a))$
  by *simp*

lemma (**in** *blockDAG*) *genesis-existAlt*:
  ∃ *a. is-genesis-node a*
  using *genesis genesisAlt*
  by (*metis reachable1-reachable reachable-refl*)

lemma (**in** *blockDAG*) *unique-genesis*: *is-genesis-node a* ∧ *is-genesis-node b* ⟶ *a* = *b*
  using *genesisAlt reachable-trans cycle-free*
    reachable-refl reachable-reachable1-trans reachable-neq-reachable1
  by (*metis* (*full-types*))

lemma (**in** *blockDAG*) *genesis-unique-exists*:
  ∃!*a. is-genesis-node a*
  using *genesis-existAlt unique-genesis* **by** *auto*

lemma (**in** *blockDAG*) *genesis-in-verts*:
genesis-node ∈ *verts G*
using *is-genesis-node.simps genesis-node-def genesis-existAlt the1I2 genesis-unique-exists*
by *metis*

lemma (**in** *blockDAG*) *genesis-reaches-nothing*:
assumes $a \rightarrow^+ b$
shows ¬ *is-genesis-node a*
using *is-genesis-node.simps genesis-node-def genesis-existAlt cycle-free*
reachable1-reachable-trans assms reachable1-in-verts(*2*)
by (*metis*)

3.2.2 Tips

lemma (**in** *blockDAG*) *tips-exist*:
∃ *x*. *is-tip G x*
unfolding *is-tip.simps*
proof (*rule ccontr*)
assume ∄ *x*. *x* ∈ *verts G* ∧(∀ *xa*∈*verts G*. (*xa*, *x*) ∉ (*arcs-ends G*)$^+$)
then have *contr*: ∀ *x*. *x* ∈ *verts G* ⟶ (∃ *y*. $y \rightarrow^+ x$)
by *auto*
have ∀ *x y*. $y \rightarrow^+ x$ ⟶ {*z*. *x* $\rightarrow^+$ *z*} ⊆ {*z*. *y* $\rightarrow^+$ *z*}
using *Collect-mono trancl-trans*
by *metis*
then have *sub*: ∀ *x y*. $y \rightarrow^+ x$ ⟶ {*z*. *x* $\rightarrow^+$ *z*} ⊂ {*z*. *y* $\rightarrow^+$ *z*}
using *cycle-free* **by** *auto*
have *part*: ∀ *x*. {*z*. *x* $\rightarrow^+$ *z*} ⊆ *verts G*
using *reachable1-in-verts* **by** *auto*
then have *fin*: ∀ *x*. *finite* {*z*. *x* $\rightarrow^+$ *z*}
using *finite-verts finite-subset*
by *metis*
then have *trans*: ∀ *x y*. $y \rightarrow^+ x$ ⟶ *card* {*z*. *x* $\rightarrow^+$ *z*} < *card* {*z*. *y* $\rightarrow^+$ *z*}
using *sub psubset-card-mono* **by** *metis*
then have *inf*: ∀ *y* ∈ *verts G*. ∃ *x*. *card* {*z*. *x* $\rightarrow^+$ *z*} > *card* {*z*. *y* $\rightarrow^+$ *z*}
using *fin contr genesis*
reachable1-in-verts(*1*)
by (*metis* (*mono-tags*, *lifting*))
have *all*: ∀ *k*. ∃ *x* ∈ *verts G*. *card* {*z*. *x* $\rightarrow^+$ *z*} > *k*
proof
fix *k*
show ∃ *x* ∈ *verts G*. *k* < *card* {*z*. *x* $\rightarrow^+$ *z*}
proof(*induct k*)
case *0*
then show *?case*
using *inf neq0-conv*
by (*metis contr genesis-in-verts local.trans reachable1-in-verts*(*1*))
next
case (*Suc k*)
then show *?case*

```
      using Suc-lessI inf
      by (metis contr local.trans reachable1-in-verts(1))
    qed
  qed
  then have less: ∃ x ∈ verts G.  card (verts G) < card {z. x →⁺ z} by simp
  also
  have ∀ x. card  {z. x →⁺ z} ≤ card (verts G)
    using fin part finite-verts not-le
    by (simp add: card-mono)
  then show False
    using less not-le by auto
qed

lemma (in blockDAG) tips-not-empty:
  shows tips G ≠ {}
proof(rule ccontr)
  assume as1: ¬ tips G ≠ {}
  obtain t where t-in: is-tip G t using tips-exist by auto
  then have t-inV: t ∈ verts G by auto
  then have t ∈ tips G using tips-def CollectI t-in by metis
  then show False using as1 by auto
qed


lemma (in blockDAG) reached-by:
  assumes v ∉ tips G
    and  v ∈ verts G
  shows ∃ t ∈ verts G. t →⁺ v
  using assms
  unfolding tips-def is-tip.simps
  by auto

lemma (in blockDAG) reached-by-tip:
  assumes v ∉ tips G
    and  v ∈ verts G
  shows ∃ t ∈ tips G. t →⁺ v
proof(rule ccontr)
  assume as1: ¬ (∃ t∈tips G. t →⁺ v)
  then have ∀ w.  w →⁺ v ⟶ ¬ is-tip G w
    unfolding tips-def using reachable1-in-verts(1) CollectI
    by blast
  then have contr: ∀ w.  w →⁺ v ⟶  (∃ y. y→⁺w ∧  y→⁺v)
    using as1 reachable1-in-verts(1) reached-by trancl-trans
    by metis
  have ∀  x y. y→⁺x ⟶  {z. x →⁺ z} ⊆ {z. y →⁺ z}
    using  Collect-mono trancl-trans
    by metis
  then have sub: ∀  x y. y→⁺x ⟶  {z. x →⁺ z} ⊂ {z. y →⁺ z}
    using cycle-free by auto
```

```
  have part: ∀ x. {z. x →⁺ z} ⊆ verts G
    using reachable1-in-verts by auto
  then have fin: ∀ x. finite {z. x →⁺ z}
    using finite-verts finite-subset
    by metis
  then have trans: ∀ x y. y→⁺x ⟶ card {z. x →⁺ z} < card {z. y →⁺ z}
    using sub psubset-card-mono by metis
  then have inf: ∀ w. w →⁺ v ⟶ (∃ x. x →⁺ v ∧ card {z. x →⁺ z} > card {z.
w →⁺ z})
    using fin contr genesis
      reachable1-in-verts(1)
    by metis
  have all: ∀ k. ∃ w ∈ verts G. w→⁺v ∧ card {z. w →⁺ z} > k
  proof
    fix k
    show ∃ w ∈ verts G. w→⁺v ∧ card {z. w →⁺ z} > k
    proof(induct k )
      case 0
      then show ?case
        using inf neq0-conv assms(1) assms(2) local.trans reached-by contr
        by (metis less-nat-zero-code)
    next
      case (Suc k)
      then show ?case
        using Suc-lessI inf reachable1-in-verts(1)
        by (metis)
    qed
  qed
  then have less: ∃ x ∈ verts G. card (verts G) < card {z. x →⁺ z}
    by blast
  also
  have ∀ x. card {z. x →⁺ z} ≤ card (verts G)
    using fin part finite-verts not-le
    by (simp add: card-mono)
  then show False
    using less not-le by auto
qed

lemma (in blockDAG) tips-unequal-gen:
  assumes card( verts G) > 1
    and is-tip G p
  shows ¬ is-genesis-node p
proof (rule ccontr)
  assume as: ¬ ¬ is-genesis-node p
  have b1: 1 < card (verts G) using assms by linarith
  then have 0 < card ((verts G) − {p}) using card-Suc-Diff1 as finite-verts b1
by auto
  then have ((verts G) − {p}) ≠ {} using card-gt-0-iff by blast
  then obtain y where y-def:y ∈ (verts G) − {p} by auto
```

```
  then have uneq: y ≠ p by auto
  then have reachable1 G y p using is-genesis-node.simps as
      reachable-neq-reachable1 Diff-iff y-def
    by metis
  then have ¬ is-tip G p
    by (meson is-tip.elims(2) reachable1-in-verts(1))
  then show False using assms by simp
qed



lemma (in blockDAG) tips-unequal-gen-exist:
  assumes card( verts G) > 1
  shows ∃ p. p ∈ verts G ∧ is-tip G p ∧ ¬is-genesis-node p
proof −
  have b1: 1 < card (verts G) using assms by linarith
  obtain x where x-in: x ∈ (verts G) ∧ is-genesis-node x
    using genesis genesisAlt genesis-node-def by blast
  then have 0 < card ((verts G) − {x}) using card-Suc-Diff1 x-in finite-verts b1
by auto
  then have ((verts G) − {x}) ≠ {} using card-gt-0-iff by blast
  then obtain y where y-def:y ∈ (verts G) − {x} by auto
  then have uneq: y ≠ x by auto
  have y-in: y ∈ (verts G) using y-def by simp
  then have reachable1 G y x using is-genesis-node.simps x-in
      reachable-neq-reachable1 uneq by simp
  then have ¬ is-tip G x
    by (meson is-tip.elims(2) y-in)
  then obtain z where z-def: z ∈ (verts G) − {x} ∧ is-tip G z using tips-exist
      is-tip.simps by auto
  then have uneq: z ≠ x by auto
  have z-in: z ∈ verts G using z-def by simp
  have ¬ is-genesis-node z
  proof (rule ccontr, safe)
    assume is-genesis-node z
    then have x = z using unique-genesis x-in by auto
    then show False using uneq by simp
  qed
  then show ?thesis using z-def by auto
qed

lemma (in blockDAG)  del-tips-bDAG:
  assumes is-tip G t
    and  ¬is-genesis-node t
  shows blockDAG (del-vert t)
  unfolding blockDAG-def blockDAG-axioms-def
proof safe
  show DAG(del-vert t)
    using del-tips-dag assms by simp
```

next
fix *u v e*
assume *wf-digraph.arc* (*del-vert t*) *e* (*u*, *v*)
then have *arc*: *arc e* (*u*,*v*) **using** *del-vert-simps wf-digraph.arc-def arc-def*
by (*metis* (*no-types*, *lifting*) *mem-Collect-eq wf-digraph-del-vert*)
assume $u \rightarrow^+_{\textit{pre-digraph.del-arc}\ (\textit{del-vert}\ t)\ e}\ v$
then have *path*: $u \rightarrow^+_{\textit{del-arc}\ e}\ v$
using *del-arc-subgraph subgraph-del-vert digraph-axioms*
digraph-subgraph
by (*metis arcs-ends-mono trancl-mono*)
show *False* **using** *arc path only-new* **by** *simp*
next
obtain *g* **where** *gen*: *is-genesis-node g* **using** *genesisAlt genesis* **by** *auto*
then have *genp*: *g* ∈ *verts* (*del-vert t*)
using *assms*(*2*) *genesis del-vert-simps* **by** *auto*
have (∀ *r*. *r* ∈ *verts* (*del-vert t*) ⟶ $r \rightarrow^*_{\textit{del-vert}\ t}\ g$)
proof *safe*
fix *r*
assume *in-del*: *r* ∈ *verts* (*del-vert t*)
then obtain *p* **where** *path*: *awalk r p g*
using *reachable-awalk is-genesis-node.simps del-vert-simps gen* **by** *auto*
have *no-head*: *t* ∉ (*set* (*map* (λ*s*. (*head G s*)) *p*))
proof (*rule ccontr*)
assume ¬ *t* ∉ (*set* (*map* (λ*s*. (*head G s*)) *p*))
then have *as*: *t* ∈ (*set* (*map* (λ*s*. (*head G s*)) *p*))
by *auto*
then obtain *e* **where** *tl*: *t* = (*head G e*) ∧ *e* ∈ *arcs G*
using *wf-digraph-def awalk-def path* **by** *auto*
then obtain *u* **where** *hd*: *u* = (*tail G e*) ∧ *u* ∈ *verts G*
using *wf-digraph-def tl* **by** *auto*
have *t* ∈ *verts G*
using *assms*(*1*) *is-tip.simps* **by** *auto*
then have *arc-to-ends G e* = (*u*, *t*) **using** *tl*
by (*simp add*: *arc-to-ends-def hd*)
then have *reachable1 G u t*
using *dominatesI tl* **by** *blast*
then show *False*
using *is-tip.simps assms*(*1*)
hd **by** *auto*
qed
have *neither*: *r* ≠ *t* ∧ *g* ≠ *t*
using *del-vert-def assms*(*2*) *gen in-del* **by** *auto*
have *no-tail*: *t* ∉ (*set* (*map* (*tail G*) *p*))
proof(*rule ccontr*)
assume *as2*: ¬ *t* ∉ *set* (*map* (*tail G*) *p*)
then have *tl2*: *t* ∈ *set* (*map* (*tail G*) *p*) **by** *auto*
then have *t* ∈ *set* (*map* (*head G*) *p*)
proof (*induct rule*: *cas.induct*)
case (*1 u v*)

```
    then have v ∉ set (map (tail G) []) by auto
    then show v ∈ set (map (tail G) []) ⟹ v ∈ set (map (head G) [])
      by auto
  next
    case (2 u e es v)
    then show ?case
      using set-awalk-verts-not-Nil-cas neither awalk-def cas.simps(2) path
      by (metis UnCI tl2 awalk-verts-conv'
          cas-simp list.simps(8) no-head set-ConsD)
  qed
  then show False using no-head by auto
qed
have pre-digraph.awalk (del-vert t) r p g
  unfolding pre-digraph.awalk-def
proof safe
  show r ∈ verts (del-vert t) using in-del by simp
next
  fix x
  assume as3: x ∈ set p
  then have ht: head G x ≠ t ∧ tail G x ≠ t
    using no-head no-tail by auto
  have  x ∈ arcs G
    using awalk-def path subsetD as3 by auto
  then show x ∈ arcs (del-vert t) using del-vert-simps(2) ht by auto
next
  have pre-digraph.cas G r p g using path by auto
  then show pre-digraph.cas (del-vert t) r p g
  proof(induct p arbitrary:r)
    case Nil
    then have r = g using awalk-def cas.simps by auto
    then show ?case using pre-digraph.cas.simps(1)
      by (metis)
  next
    case (Cons a p)
    assume pre: ⋀r. (cas r p g ⟹ pre-digraph.cas (del-vert t) r p g)
      and one: cas r (a # p) g
    then have two: cas (head G a) p g
      using awalk-def  by auto
    then have t: tail (del-vert t) a = r
      using one cas.simps awalk-def del-vert-simps(3) by auto
    then show ?case
      unfolding pre-digraph.cas.simps(2) t
      using pre two del-vert-simps(4) by auto
  qed
qed
then show r →*del-vert t g by (meson wf-digraph.reachable-awalkI
      del-tips-dag assms(1) DAG-def digraph-def fin-digraph-def)
qed
then show  ∃ p ∈ verts (del-vert t) .
```

$(\forall r.\ r \in verts\ (del\text{-}vert\ t) \longrightarrow (r \rightarrow^+_{del\text{-}vert\ t} p \vee r = p))$
using *gen genp*
by (*metis reachable-rtranclI rtranclD*)
qed

lemma (**in** *blockDAG*) *tips-cases* [*consumes 2*, *case-names ma past nma*]:
assumes $p \in tips\ G$
and $x \in verts\ G$
obtains (*ma*) $x = p$
| (*past*) $x \in past\text{-}nodes\ G\ p$
| (*nma*) $x \in anticone\ G\ p$
proof −
consider (*eq*)$x = p$ | (*neq*) $\neg x = p$ **by** *auto*
then show *?thesis*
proof(*cases*)
case *eq*
then show *thesis* **using** *eq ma* **by** *simp*
next
case *neq*
consider (*in-p*)$x \in past\text{-}nodes\ G\ p$ | (*nin-p*)$x \notin past\text{-}nodes\ G\ p$ **by** *auto*
then show *?thesis*
proof(*cases*)
case *in-p*
then show *?thesis* **using** *past* **by** *auto*
next
case *nin-p*
then have *nn*: $\neg\ p \rightarrow^+_G x$ **using** *nin-p past-nodes.simps assms*(*2*) **by** *auto*
have $\neg\ x \rightarrow^+_G p$ **using** *is-tip.simps assms tips-def CollectD* **by** *metis*
then have $x \in anticone\ G\ p$ **using** *anticone.simps neq nn assms*(*2*) **by** *auto*
then show *?thesis* **using** *nma* **by** *auto*
qed
qed
qed

3.3 Future Nodes

lemma (**in** *blockDAG*) *future-nodes-ex*:
assumes $a \in verts\ G$
shows $a \notin future\text{-}nodes\ G\ a$
using *cycle-free future-nodes.simps reachable-def* **by** *auto*

3.3.1 Reduce Past

lemma (**in** *blockDAG*) *reduce-past-not-empty*:
assumes $a \in verts\ G$
and $\neg is\text{-}genesis\text{-}node\ a$
shows $(verts\ (reduce\text{-}past\ G\ a)) \neq \{\}$
proof −
obtain *g*

```
    where gen: is-genesis-node g using genesis-existAlt by auto
  have ex: g ∈ verts (reduce-past G a) using reduce-past.simps past-nodes.simps
       genesisAlt reachable-neq-reachable1 reachable-reachable1-trans gen assms(1)
assms(2) by auto
 then show (verts (reduce-past G a)) ≠ {} using ex by auto
qed

lemma (in blockDAG) reduce-less:
  assumes a ∈ verts G
  shows card (verts (reduce-past G a)) < card (verts G)
proof −
  have past-nodes G a ⊂ verts G
    using assms(1) past-nodes-not-refl past-nodes-verts by blast
  then show ?thesis
    by (simp add: psubset-card-mono)
qed

lemma (in blockDAG) reduce-past-dagbased:
  assumes  a ∈ verts G
    and ¬is-genesis-node a
  shows blockDAG (reduce-past G a)
  unfolding blockDAG-def DAG-def blockDAG-def

proof safe
  show digraph (reduce-past G a)
    using digraphI-induced reduce-past-induced-subgraph by auto
next
  show DAG-axioms (reduce-past G a)
    unfolding DAG-axioms-def
    using cycle-free reduce-past-path by metis
next
  show blockDAG-axioms (reduce-past G a)
    unfolding blockDAG-axioms-def
  proof safe
    fix u v e
    assume arc: wf-digraph.arc (reduce-past G a) e (u, v)
    then show  u →⁺pre-digraph.del-arc (reduce-past G a) e v ⟹ False
    proof −
      assume e-in: (wf-digraph.arc (reduce-past G a) e (u, v))
      then have (wf-digraph.arc G e (u, v))
        using assms reduce-past-arcs2 induced-subgraph-def arc-def
      proof −
        have wf-digraph (reduce-past G a)
       using reduce-past.simps subgraph-def subgraph-refl wf-digraph.wellformed-induce-subgraph
```

```
      by metis
    then have e ∈ arcs (reduce-past G a) ∧ tail (reduce-past G a) e = u
            ∧ head (reduce-past G a) e = v
      using  arc wf-digraph.arcE
      by metis
    then show ?thesis
      using arc-def reduce-past.simps by auto
  qed
  then have ¬ u →⁺ del-arc e v
    using only-new by auto
  then show u →⁺ pre-digraph.del-arc (reduce-past G a) e v ⟹ False
    using DAG.past-nodes-verts reduce-past.simps blockDAG-axioms subs
      del-arc-subgraph digraph.digraph-subgraph digraph-axioms
      subgraph-induce-subgraphI
    by (metis arcs-ends-mono trancl-mono)
 qed
next
 obtain p where gen: is-genesis-node p using genesis-existAlt by auto
 have pe: p ∈ verts (reduce-past G a) ∧ (∀ r. r ∈ verts (reduce-past G a) ⟶ r
→⁺ reduce-past G a p)
 proof
 show p ∈ verts (reduce-past G a) using genesisAlt induce-reachable-preserves-paths
   reduce-past.simps past-nodes.simps reachable1-reachable induce-subgraph-verts
assms(1)
     assms(2) gen mem-Collect-eq reachable-neq-reachable1
    by (metis (no-types, lifting))

 next
  show ∀ r. r ∈ verts (reduce-past G a) ⟶ r →⁺ reduce-past G a p
  proof safe
    fix r a
    assume in-past: r ∈ verts (reduce-past G a)
    then have con: r →⁺ p using gen genesisAlt past-nodes-verts by auto
    then show r →⁺ reduce-past G a p
    proof −
      have f1: r ∈ verts G ∧ a →⁺ r
        using in-past past-nodes-verts by force
      obtain aaa :: 'a set ⇒ 'a set ⇒ 'a where
        f2: ∀ x0 x1. (∃ v2. v2 ∈ x1 ∧ v2 ∉ x0) = (aaa x0 x1 ∈ x1 ∧ aaa x0 x1 ∉
x0)
        by moura
      have r →⁺ aaa (past-nodes G a) (Collect (reachable G r))
            ⟶ a →⁺ aaa (past-nodes G a) (Collect (reachable G r))
        using f1 by (meson reachable1-reachable-trans)
       then have aaa (past-nodes G a) (Collect (reachable G r)) ∉ Collect
(reachable G r)
                  ∨ aaa (past-nodes G a) (Collect (reachable G r)) ∈ past-nodes
G a
        by (simp add: reachable-in-verts(2))
```

```
        then have Collect (reachable G r) ⊆ past-nodes G a
          using f2 by (metis subsetI)
        then show ?thesis
             using con  induce-reachable-preserves-paths reachable-induce-ss re-
duce-past.simps
          by (metis (no-types))
       qed
      qed
    qed
    show
      ∃ p ∈ verts (reduce-past G a). (∀ r. r ∈ verts (reduce-past G a)
        ⟶ (r →⁺reduce-past G a p ∨ r = p))
      using pe
      by (metis reachable-rtranclI rtranclD)
  qed
qed



lemma (in blockDAG) reduce-past-gen:
  assumes ¬is-genesis-node a
    and a ∈ verts G
  shows blockDAG.is-genesis-node G b ⟹ blockDAG.is-genesis-node (reduce-past
G a) b
proof −
  assume gen: blockDAG.is-genesis-node G b
  have une: b ≠ a  using gen assms(1) genesis-unique-exists by auto
  have a →* b using gen assms(2) by simp
  then have a →⁺ b
    using reachable-neq-reachable1 is-genesis-node.simps assms(2) une by auto
  then have b ∈ (past-nodes G  a) using past-nodes.simps gen by auto
 then have inv: b ∈ verts (reduce-past G a) using reduce-past.simps induce-subgraph-verts

    by auto
  have∀ r. r ∈ verts (reduce-past G a) ⟶ r →*reduce-past G a b
  proof safe
    fix r a
    assume in-past: r ∈ verts (reduce-past G a)
    then have con: r →* b using gen genesisAlt past-nodes-verts by auto
    then show r →*reduce-past G a b
    proof −
      have f1: r ∈ verts G ∧ a →⁺ r
        using in-past past-nodes-verts by force
      obtain aaa :: 'a set ⇒ 'a set ⇒ 'a where
        f2: ∀ x0 x1. (∃ v2. v2 ∈ x1 ∧ v2 ∉ x0) = (aaa x0 x1 ∈ x1 ∧ aaa x0 x1 ∉ x0)
        by moura
      have r →* aaa (past-nodes G a) (Collect (reachable G r))
           ⟶ a →⁺ aaa (past-nodes G a) (Collect (reachable G r))
        using f1 by (meson reachable1-reachable-trans)
```

then have *aaa* (*past-nodes G a*) (*Collect* (*reachable G r*)) ∉ *Collect* (*reachable G r*)
∨ *aaa* (*past-nodes G a*) (*Collect* (*reachable G r*)) ∈ *past-nodes G a*
by (*simp add*: *reachable-in-verts*(*2*))
then have *Collect* (*reachable G r*) ⊆ *past-nodes G a*
using *f2* **by** (*meson subsetI*)
then show *?thesis*
using *con induce-reachable-preserves-paths reachable-induce-ss reduce-past.simps*
by (*metis* (*no-types*))
qed
qed
then show *blockDAG.is-genesis-node* (*reduce-past G a*) *b* **using** *inv is-genesis-node.simps*
by (*metis assms*(*1*) *assms*(*2*) *blockDAG.is-genesis-node.elims*(*3*)
reduce-past-dagbased)
qed

lemma (**in** *blockDAG*) *reduce-past-gen-rev*:
assumes ¬*is-genesis-node a*
and *a* ∈ *verts G*
shows *blockDAG.is-genesis-node* (*reduce-past G a*) *b* ⟹ *blockDAG.is-genesis-node G b*
proof −
assume *as1*: *blockDAG.is-genesis-node* (*reduce-past G a*) *b*
have *bD*: *blockDAG* (*reduce-past G a*) **using** *assms reduce-past-dagbased blockDAG-axioms* **by** *simp*
obtain *gen* **where** *is-gen*: *is-genesis-node gen* **using** *genesis-unique-exists* **by** *auto*
then have *blockDAG.is-genesis-node* (*reduce-past G a*) *gen* **using** *reduce-past-gen assms* **by** *auto*
then have *gen* = *b* **using** *as1 blockDAG.unique-genesis bD* **by** *metis*
then show *blockDAG.is-genesis-node* (*reduce-past G a*) *b* ⟹ *blockDAG.is-genesis-node G b*
using *is-gen* **by** *auto*
qed

lemma (**in** *blockDAG*) *reduce-past-gen-eq*:
assumes ¬*is-genesis-node a*
and *a* ∈ *verts G*
shows *blockDAG.is-genesis-node* (*reduce-past G a*) *b* = *blockDAG.is-genesis-node G b*
using *reduce-past-gen reduce-past-gen-rev assms assms* **by** *metis*

3.3.2 Reduce Past Reflexiv

lemma (**in** *blockDAG*) *reduce-past-refl-induced-subgraph*:
shows *induced-subgraph* (*reduce-past-refl G a*) *G*
using *induced-induce past-nodes-refl-verts* **by** *auto*

lemma (**in** *blockDAG*) *reduce-past-refl-arcs2*:
e ∈ *arcs* (*reduce-past-refl G a*) ⟹ *e* ∈ *arcs G*
using *reduce-past-arcs* **by** *auto*

lemma (**in** *blockDAG*) *reduce-past-refl-digraph*:
assumes *a* ∈ *verts G*
shows *digraph* (*reduce-past-refl G a*)
using *digraphI-induced reduce-past-refl-induced-subgraph reachable-mono* **by** *simp*

lemma (**in** *blockDAG*) *reduce-past-refl-dagbased*:
assumes *a* ∈ *verts G*
shows *blockDAG* (*reduce-past-refl G a*)
unfolding *blockDAG-def DAG-def*
proof *safe*
show *digraph* (*reduce-past-refl G a*)
using *reduce-past-refl-digraph assms*(*1*) **by** *simp*
next
show *DAG-axioms* (*reduce-past-refl G a*)
unfolding *DAG-axioms-def*
using *cycle-free reduce-past-refl-induced-subgraph reachable-mono*
by (*meson arcs-ends-mono induced-subgraph-altdef trancl-mono*)
next
show *blockDAG-axioms* (*reduce-past-refl G a*)
unfolding *blockDAG-axioms*
proof
fix *u v*
show ∀ *e*. *u* $\rightarrow^+_{\textit{pre-digraph.del-arc}\ (\textit{reduce-past-refl}\ G\ a)\ e}$ *v*
⟶ ¬ *wf-digraph.arc* (*reduce-past-refl G a*) *e* (*u*, *v*)
proof *safe*
fix *e*
assume *a*: *wf-digraph.arc* (*reduce-past-refl G a*) *e* (*u*, *v*)
and *b*: *u* $\rightarrow^+_{\textit{pre-digraph.del-arc}\ (\textit{reduce-past-refl}\ G\ a)\ e}$ *v*
have *edge*: *wf-digraph.arc G e* (*u*, *v*)
using *assms reduce-past-arcs2 induced-subgraph-def arc-def*
proof −
have *wf-digraph* (*reduce-past-refl G a*)
using *reduce-past-refl-digraph digraph-def* **by** *auto*
then have *e* ∈ *arcs* (*reduce-past-refl G a*) ∧ *tail* (*reduce-past-refl G a*) *e* = *u*
∧ *head* (*reduce-past-refl G a*) *e* = *v*
using *wf-digraph.arcE arc-def a*
by (*metis* (*no-types*))
then show *arc e* (*u*, *v*)
using *arc-def reduce-past-refl.simps* **by** *auto*
qed
have *u* $\rightarrow^+_{\textit{pre-digraph.del-arc}\ G\ e}$ *v*
using *a b reduce-past-refl-digraph del-arc-subgraph digraph-axioms*
digraphI-induced past-nodes-refl-verts reduce-past-refl.simps
reduce-past-refl-induced-subgraph subgraph-induce-subgraphI arcs-ends-mono

trancl-mono
by *metis*
then show *False*
using *edge only-new* **by** *simp*
qed
next
obtain *p* **where** *gen*: *is-genesis-node p* **using** *genesis-existAlt* **by** *auto*
have *pe*: $p \in$ *verts* (*reduce-past-refl G a*)
using *genesisAlt induce-reachable-preserves-paths*
reduce-past.simps past-nodes.simps reachable1-reachable induce-subgraph-verts
gen mem-Collect-eq reachable-neq-reachable1
assms **by** *force*
have *reaches*: $(\forall r.\ r \in$ *verts* (*reduce-past-refl G a*) $\longrightarrow$
$(r \rightarrow^+_{\textit{reduce-past-refl G a}} p \vee r = p))$
proof *safe*
fix *r*
assume *in-past*: $r \in$ *verts* (*reduce-past-refl G a*)
assume *une*: $r \neq p$
then have *con*: $r \rightarrow^* p$ **using** *gen genesisAlt reachable-in-verts*
reachable1-reachable
by (*metis in-past induce-subgraph-verts*
past-nodes-refl-verts reduce-past-refl.simps subsetD)
have $a \rightarrow^* r$ **using** *in-past* **by** *auto*
then have *reach*: $r \rightarrow^*_{G \restriction \{w.\ a \rightarrow^* w\}} p$
proof(*induction*)
case *base*
then show *?case*
using *con induce-reachable-preserves-paths*
by (*metis*)
next
case (*step x y*)
then show *?case*
proof −
have *Collect* (*reachable G y*) $\subseteq$ *Collect* (*reachable G x*)
using *adj-reachable-trans step.hyps*(*1*) **by** *force*
then show *?thesis*
using *reachable-induce-ss step.IH reachable-neq-reachable1*
by *metis*
qed
qed
then show $r \rightarrow^+_{\textit{reduce-past-refl G a}} p$ **unfolding** *reduce-past-refl.simps*
past-nodes-refl.simps **using** *reachable-in-verts une wf-digraph.reachable-neq-reachable1*
by (*metis* (*mono-tags*, *lifting*) *Collect-cong wellformed-induce-subgraph*)
qed
then show $\exists p \in$ *verts* (*reduce-past-refl G a*). $(\forall r.\ r \in$ *verts* (*reduce-past-refl G a*)
$\longrightarrow (r \rightarrow^+_{\textit{reduce-past-refl G a}} p \vee r = p))$ **unfolding** *blockDAG-axioms-def*
using *pe reaches* **by** *auto*
qed

qed

3.3.3 Genesis Graph

definition (**in** *blockDAG*) *gen-graph*::('*a*,'*b*) *pre-digraph* **where**
gen-graph = *induce-subgraph G* {*blockDAG.genesis-node G*}

lemma (**in** *blockDAG*) *gen-gen* :*verts* (*gen-graph*) = {*genesis-node*}
unfolding *genesis-node-def gen-graph-def* **by** *simp*

lemma (**in** *blockDAG*) *gen-graph-one*: *card* (*verts gen-graph*) = *1* **using** *gen-gen*
by *simp*

lemma (**in** *blockDAG*) *gen-graph-digraph*:
digraph gen-graph
using *digraphI-induced induced-induce gen-graph-def*
genesis-in-verts **by** *simp*

lemma (**in** *blockDAG*) *gen-graph-empty-arcs*:
arcs gen-graph = {}
proof(*rule ccontr*)
assume ¬ *arcs gen-graph* = {}
then have *ex*: ∃ *a*. *a* ∈ (*arcs gen-graph*)
by *blast*
also have ∀ *a*. *a* ∈ (*arcs gen-graph*)⟶ *tail G a* = *head G a*
proof *safe*
fix *a*
assume *a* ∈ *arcs gen-graph*
then show *tail G a* = *head G a*
using *digraph-def induced-subgraph-def induce-subgraph-verts*
induced-induce gen-graph-def **by** *simp*
qed
then show *False*
using *digraph-def ex gen-graph-def gen-graph-digraph induce-subgraph-head induce-subgraph-tail*
loopfree-digraph.no-loops
by *metis*
qed

lemma (**in** *blockDAG*) *gen-graph-sound*:
blockDAG (*gen-graph*)
unfolding *blockDAG-def DAG-def blockDAG-axioms-def*
proof *safe*
show *digraph gen-graph* **using** *gen-graph-digraph* **by** *simp*
next
have (*arcs-ends gen-graph*)$^+$ = {}
using *trancl-empty gen-graph-empty-arcs* **by** (*simp add*: *arcs-ends-def*)
then show *DAG-axioms gen-graph*

by (*simp add*: *DAG-axioms.intro*)
next
fix *u v e*
have *wf-digraph.arc gen-graph e* (*u*, *v*) ≡ *False*
using *wf-digraph.arc-def gen-graph-empty-arcs*
by (*simp add*: *wf-digraph.arc-def wf-digraph-def*)
then show *wf-digraph.arc gen-graph e* (*u*, *v*) ⟹
u $\rightarrow^+_{pre\text{-}digraph.del\text{-}arc\ gen\text{-}graph\ e}$ *v* ⟹ *False*
by *simp*
next
have *refl*: *genesis-node* $\rightarrow^*_{gen\text{-}graph}$ *genesis-node*
using *gen-gen rtrancl-on-refl*
by (*simp add*: *reachable-def*)
have ∀ *r*. *r* ∈ *verts gen-graph* ⟶ *r* $\rightarrow^*_{gen\text{-}graph}$ *genesis-node*
proof *safe*
fix *r*
assume *r* ∈ *verts gen-graph*
then have *r* = *genesis-node*
using *gen-gen* **by** *auto*
then show *r* $\rightarrow^*_{gen\text{-}graph}$ *genesis-node*
by (*simp add*: *local.refl*)
qed
then show ∃ *p* ∈ *verts gen-graph*.
(∀ *r*. *r* ∈ *verts gen-graph* ⟶ *r* $\rightarrow^+_{gen\text{-}graph}$ *p* ∨ *r* = *p*)
by (*simp add*: *gen-gen*)
qed

lemma (**in** *blockDAG*) *no-empty-blockDAG*:
shows *card* (*verts G*) > *0*
proof −
have ∃ *p*. *p* ∈ *verts G*
using *genesis-in-verts* **by** *auto*
then show *card* (*verts G*) > *0*
using *card-gt-0-iff finite-verts* **by** *blast*
qed

lemma (**in** *blockDAG*) *gen-graph-all-one*:
card (*verts* (*G*)) = *1* ⟷ *G* = *gen-graph*
using *card-1-singletonE gen-graph-def genesis-in-verts*
induce-eq-iff-induced induced-subgraph-refl singletonD gen-graph-def genesis-node-def
by (*metis gen-gen genesis-existAlt is-genesis-node.simps less-one linorder-neqE-nat*
neq0-conv no-empty-blockDAG tips-unequal-gen-exist)

lemma *blockDAG-nat-induct*[*consumes 1*, *case-names base step*]:
assumes
bD: *blockDAG Z*
and
cases: ⋀*V*. (*blockDAG V* ⟹ *card* (*verts V*) = *1* ⟹ *P V*)
⋀*W c*. (⋀*V*. (*blockDAG V* ⟹ *card* (*verts V*) = *c* ⟹ *P V*))

```
  ⟹ (blockDAG W ⟹ card (verts W) = Suc c ⟹ P W)
  shows P Z
proof −
  have bG: card (verts Z) > 0 using bD blockDAG.no-empty-blockDAG by auto
  show ?thesis
    using bG bD
  proof (induction card (verts Z)  arbitrary: Z rule: Nat.nat-induct-non-zero)
    case 1
    then show ?case using cases(1) by auto
  next
    case su: (Suc n)
    show ?case
      by (metis local.cases(2) su.hyps(2) su.hyps(3) su.prems)
  qed
qed


lemma blockDAG-nat-less-induct[consumes 1, case-names base step]:
  assumes
    bD: blockDAG Z
    and
    cases: ⋀V. (blockDAG V ⟹ card (verts V) = 1 ⟹ P V)
    ⋀W c. (⋀V. (blockDAG V ⟹ card (verts V) < c ⟹ P V))
  ⟹ (blockDAG W ⟹ card (verts W) = c ⟹ P W)
  shows P Z
proof −
  have bG: card (verts Z) > 0 using blockDAG.no-empty-blockDAG assms(1) by
auto
  show P Z
    using bD bG
  proof (induction card (verts Z) arbitrary: Z rule: less-induct)
    fix Z::('a, 'b) pre-digraph
    assume a:
     (⋀Za. card (verts Za) < card (verts Z) ⟹ blockDAG Za ⟹ 0 < card (verts
Za) ⟹ P Za)
    assume blockDAG Z
    then show P Z using a cases
      by (metis blockDAG.no-empty-blockDAG)
  qed
qed

lemma (in blockDAG) blockDAG-size-cases:
  obtains (one) card (verts G) = 1
  | (more) card (verts G) > 1
  using no-empty-blockDAG
  by linarith

lemma (in blockDAG) blockDAG-cases-one:
  shows card (verts G) = 1 ⟶ (G = gen-graph)
```

proof (*safe*)
assume *one*: *card* (*verts G*) = *1*
then have *blockDAG.genesis-node G* ∈ *verts G*
by (*simp add*: *genesis-in-verts*)
then have *only*: *verts G* = {*blockDAG.genesis-node G*}
by (*metis one card-1-singletonE insert-absorb singleton-insert-inj-eq′*)
then have *verts-equal*: *verts G* = *verts* (*blockDAG.gen-graph G*)
using *blockDAG-axioms one blockDAG.gen-graph-def induce-subgraph-def induced-induce blockDAG.genesis-in-verts*
by (*simp add*: *blockDAG.gen-graph-def*)
have *arcs G* ={}
proof (*rule ccontr*)
assume *not-empty*: *arcs G* ≠{}
then obtain *z* **where** *part-of*: *z* ∈ *arcs G*
by *auto*
then have *tail*: *tail G z* ∈ *verts G*
using *wf-digraph-def blockDAG-def DAG-def digraph-def blockDAG-axioms nomulti-digraph.axioms*(*1*)
by *metis*
also have *head*: *head G z* ∈ *verts G*
by (*metis* (*no-types*) *DAG-def blockDAG-axioms blockDAG-def digraph-def nomulti-digraph.axioms*(*1*) *part-of wf-digraph-def*)
then have *tail G z* = *head G z*
using *tail only* **by** *simp*
then have ¬ *loopfree-digraph-axioms G*
unfolding *loopfree-digraph-axioms-def*
using *part-of only DAG-def digraph-def*
by *auto*
then show *False*
using *DAG-def digraph-def blockDAG-axioms blockDAG-def loopfree-digraph-def* **by** *metis*
qed
then have *arcs G* = *arcs* (*blockDAG.gen-graph G*)
by (*simp add*: *blockDAG-axioms blockDAG.gen-graph-empty-arcs*)
then show *G* = *gen-graph*
unfolding *blockDAG.gen-graph-def*
using *verts-equal blockDAG-axioms induce-subgraph-def blockDAG.gen-graph-def* **by** *fastforce*
qed

lemma (**in** *blockDAG*) *blockDAG-cases-more*:
shows *card* (*verts G*) > *1* ⟷ (∃ *b H*. (*blockDAG H* ∧ *b* ∈ *verts G* ∧ *del-vert b* = *H*))
proof *safe*
assume *card* (*verts G*) > *1*
then have *b1*: *1* < *card* (*verts G*) **using** *no-empty-blockDAG* **by** *linarith*
obtain *x* **where** *x-in*: *x* ∈ (*verts G*) ∧ *is-genesis-node x*
using *genesis genesisAlt genesis-node-def* **by** *blast*
then have *0* < *card* ((*verts G*) − {*x*}) **using** *card-Suc-Diff1 x-in finite-verts b1*

by *auto*
then have ((*verts G*) − {*x*}) ≠ {} **using** *card-gt-0-iff* **by** *blast*
then obtain *y* **where** *y-def*:*y* ∈ (*verts G*) − {*x*} **by** *auto*
then have *uneq*: *y* ≠ *x* **by** *auto*
have *y-in*: *y* ∈ (*verts G*) **using** *y-def* **by** *simp*
then have *reachable1 G y x* **using** *is-genesis-node.simps x-in*
reachable-neq-reachable1 uneq **by** *simp*
then have ¬ *is-tip G x*
using *y-in* **by** *force*
then obtain *z* **where** *z-def*: *z* ∈ (*verts G*) − {*x*} ∧ *is-tip G z* **using** *tips-exist*
is-tip.simps **by** *auto*
then have *uneq*: *z* ≠ *x* **by** *auto*
have *z-in*: *z* ∈ *verts G* **using** *z-def* **by** *simp*
have ¬ *is-genesis-node z*
proof (*rule ccontr*, *safe*)
assume *is-genesis-node z*
then have *x* = *z* **using** *unique-genesis x-in* **by** *auto*
then show *False* **using** *uneq* **by** *simp*
qed
then have *blockDAG* (*del-vert z*) **using** *del-tips-bDAG z-def* **by** *simp*
then show (∃ *b H*. *blockDAG H* ∧ *b* ∈ *verts G* ∧ *del-vert b* = *H*) **using** *z-def*
by *auto*
next
fix *b* **and** *H*::('*a*,'*b*) *pre-digraph*
assume *bD*: *blockDAG* (*del-vert b*)
assume *b-in*: *b* ∈ *verts G*
show *card* (*verts G*) > *1*
proof (*rule ccontr*)
assume ¬ *1* < *card* (*verts G*)
then have *1* = *card* (*verts G*) **using** *no-empty-blockDAG* **by** *linarith*
then have *card* (*verts* (*del-vert b*)) = *0* **using** *b-in del-vert-def* **by** *auto*
then have ¬ *blockDAG* (*del-vert b*) **using** *bD blockDAG.no-empty-blockDAG*
by (*metis less-nat-zero-code*)
then show *False* **using** *bD* **by** *simp*
qed
qed

lemma (**in** *blockDAG*) *blockDAG-cases*:
obtains (*base*) (*G* = *gen-graph*)
| (*more*) (∃ *b H*. (*blockDAG H* ∧ *b* ∈ *verts G* ∧ *del-vert b* = *H*))
using *blockDAG-cases-one blockDAG-cases-more*
blockDAG-size-cases **by** *auto*

lemma *blockDAG-induct*[*consumes 1*, *case-names base step*]:
assumes *fund*: *blockDAG G*
assumes *cases*: ⋀*V*::('*a*,'*b*) *pre-digraph*. *blockDAG V* ⟹ *P* (*blockDAG.gen-graph*
V)
⋀*H*::('*a*,'*b*) *pre-digraph*.
(⋀*b*::'*a*. *blockDAG* (*pre-digraph.del-vert H b*) ⟹ *b* ∈ *verts H* ⟹ *P*(*pre-digraph.del-vert*

```
H b))
  ⟹ (blockDAG H ⟹ P H)
  shows P G
proof(induct-tac G rule:blockDAG-nat-induct)
  show blockDAG G using assms(1) by simp
next
  fix V::('a,'b) pre-digraph
  assume bD: blockDAG V
    and card (verts V) = 1
  then have V = blockDAG.gen-graph V
    using blockDAG.blockDAG-cases-one equal-refl  by auto
  then show P V using bD cases(1)
    by metis
next
  fix c and W::('a,'b) pre-digraph
  show (⋀V. blockDAG V ⟹ card (verts V) = c ⟹ P V) ⟹
         blockDAG W ⟹ card (verts W) = Suc c ⟹ P W
  proof −
    assume ind: ⋀V. (blockDAG V ⟹ card (verts V) = c ⟹ P V)
      and bD: blockDAG W
      and  size: card (verts W) = Suc c
    have  assm2: ⋀b. blockDAG (pre-digraph.del-vert W b)
          ⟹ b ∈ verts W ⟹ P(pre-digraph.del-vert W b)
    proof −
      fix b
      assume bD2: blockDAG (pre-digraph.del-vert W b)
      assume in-verts: b ∈ verts W
      have verts (pre-digraph.del-vert W b) = verts W − {b}
        by (simp add: pre-digraph.verts-del-vert)
      then have card ( verts (pre-digraph.del-vert W b)) = c
       using in-verts fin-digraph.finite-verts bD subs fin-digraph.fin-digraph-del-vert

         size
        by (simp add: fin-digraph.finite-verts subs
            DAG.axioms assms(1) digraph.axioms)
      then show P (pre-digraph.del-vert W b) using ind bD2 by auto
    qed
    show ?thesis using cases(2)
      by (metis assm2 bD)
  qed
qed

function genesis-nodeAlt:: ('a::linorder,'b) pre-digraph ⇒ 'a
  where genesis-nodeAlt G = (if (¬ blockDAG G) then undefined else
  if (card (verts G ) = 1) then (hd (sorted-list-of-set (verts G)))
  else genesis-nodeAlt (reduce-past G ((hd (sorted-list-of-set (tips G))))))
  by auto
termination proof
  let ?R = measure ( λG. (card (verts G)))
```

```
  show wf ?R by auto
next
  fix G ::('a::linorder,'b) pre-digraph
  assume ¬ ¬ blockDAG G
  then have bD: blockDAG G by simp
  assume card (verts G) ≠ 1
  then have bG: card (verts G) > 1 using bD blockDAG.blockDAG-size-cases by
auto
  have  set (sorted-list-of-set (tips G)) = tips G
    by (simp add: bD subs tips-def fin-digraph.finite-verts)
  then have  hd (sorted-list-of-set (tips G)) ∈ tips G
    using hd-in-set bD  tips-def bG blockDAG.tips-unequal-gen-exist
      empty-iff empty-set mem-Collect-eq
    by (metis (mono-tags, lifting))
  then show (reduce-past G (hd (sorted-list-of-set (tips G))), G) ∈ measure (λG.
card (verts G))
    using blockDAG.reduce-less bD
    using tips-def by fastforce
qed

lemma genesis-nodeAlt-one-sound:
  assumes bD: blockDAG G
    and one: card (verts G) = 1
  shows blockDAG.is-genesis-node G (genesis-nodeAlt G)
proof −
  interpret B: blockDAG G using assms(1) by simp
  have exone: ∃! x. x ∈ (verts G)
    using  one B.genesis-in-verts B.genesis-unique-exists B.reduce-less
    B.reduce-past-dagbased less-nat-zero-code less-one B.gen-gen B.gen-graph-all-one
singleton-iff
    by (metis)
  then have sorted-list-of-set (verts G) ≠ []
    by (metis card.infinite card-0-eq finite.emptyI one
        sorted-list-of-set-empty sorted-list-of-set-inject zero-neq-one)
  then have genesis-nodeAlt G ∈ verts G using hd-in-set genesis-nodeAlt.simps
bD exone
    by (metis one set-sorted-list-of-set sorted-list-of-set.infinite)
  then show one-sound: B.is-genesis-node (genesis-nodeAlt G)
    using one B.blockDAG-size-cases B.reduce-less
     B.reduce-past-dagbased less-one B.genesis-unique-exists B.is-genesis-node.elims(2)
exone
    by (metis)
qed

lemma genesis-nodeAlt-sound :
  assumes blockDAG G
  shows blockDAG.is-genesis-node G (genesis-nodeAlt G)
proof(induct-tac G rule:blockDAG-nat-less-induct)
  show blockDAG G using assms by simp
```

next
fix V::$('a,'b)$ *pre-digraph*
assume *bD*: *blockDAG V*
assume *one*: *card* (*verts V*) = *1*
then show *blockDAG.is-genesis-node V* (*genesis-nodeAlt V*)
using *genesis-nodeAlt-one-sound bD*
by *blast*
next
fix W::$('a,'b)$ *pre-digraph*
fix c::*nat*
assume *basis*:
($\bigwedge V$::$('a,'b)$ *pre-digraph*. *blockDAG V* $\Longrightarrow$ *card* (*verts V*) < *c* $\Longrightarrow$
blockDAG.is-genesis-node V (*genesis-nodeAlt V*))
assume *bD*: *blockDAG W*
interpret *B*: *blockDAG W* **using** *bD* **by** *simp*
assume *cd*: *card* (*verts W*) = *c*
consider (*one*) *card* (*verts W*) = *1* | (*more*) *card* (*verts W*) > *1*
using *bD blockDAG.blockDAG-size-cases* **by** *blast*
then show *blockDAG.is-genesis-node W* (*genesis-nodeAlt W*)
proof(*cases*)
case *one*
then show *?thesis* **using** *genesis-nodeAlt-one-sound bD*
by *blast*
next
case *more*
then have *not-one*: *1* ≠ *card* (*verts W*) **by** *auto*
have *se*: *set* (*sorted-list-of-set* (*tips W*)) = *tips W*
by (*simp add*: *tips-def*)
obtain *a* **where** *a-def*: *a* = *hd* (*sorted-list-of-set* (*tips W*))
by *simp*
have *tip*: *a* ∈ *tips W*
using *se a-def hd-in-set bD tips-def more blockDAG.tips-unequal-gen-exist*
empty-iff empty-set mem-Collect-eq
by (*metis* (*mono-tags*, *lifting*))
then have *ver*: *a* ∈ *verts W*
by (*simp add*: *tips-def a-def*)
then have *card* (*verts* (*reduce-past W a*)) < *card* (*verts W*)
using *more cd blockDAG.reduce-less bD*
by *metis*
then have *cd2*: *card* (*verts* (*reduce-past W a*)) < *c*
using *cd* **by** *simp*
have *is-tip W a* **using** *tip CollectD* **unfolding** *tips-def* **by** *simp*
then have *n-gen*: ¬ *B.is-genesis-node a*
using *B.tips-unequal-gen more* **by** *simp*
then have *bD2*: *blockDAG* (*reduce-past W a*)
using *B.reduce-past-dagbased ver bD* **by** *auto*
have *ff*: *blockDAG.is-genesis-node* (*reduce-past W a*)
(*genesis-nodeAlt* (*reduce-past W a*)) **using** *cd2 basis bD2 more*
by *blast*

have *rec*:
genesis-nodeAlt W = genesis-nodeAlt (reduce-past W (hd (sorted-list-of-set (tips W))))
using *genesis-nodeAlt.simps not-one bD*
by *metis*
show *?thesis* **using** *rec ff bD n-gen ver blockDAG.reduce-past-gen-eq a-def* **by** *metis*
qed
qed

end

theory *Spectre*
imports *Main Graph-Theory.Graph-Theory blockDAG*
begin

Based on the SPECTRE paper by Sompolinsky, Lewenberg and Zohar 2016

4 Spectre

4.1 Definitions

Function to check and break occuring ties

fun *tie-break-int*:: *'a*::*linorder* ⇒ *'a* ⇒ *int* ⇒ *int*
where *tie-break-int a b i* =
(*if i=0 then* (*if* (*b* < *a*) *then* −*1 else 1*) *else i*)

Sign function with 0

fun *signum* :: *int* ⇒ *int*
where *signum a* = (*if a* > *0 then 1 else if a* < *0 then* −*1 else 0*)

Spectre core algorithm, $vote - SpectreV\,abc$ returns 1 if a votes in favour of b (or $b = c$), -1 if a votes in favour of c, 0 otherwise

function *vote-Spectre* :: (*'a*::*linorder*,*'b*) *pre-digraph* ⇒*'a* ⇒ *'a* ⇒ *'a* ⇒ *int*
where
vote-Spectre V a b c = (
if (¬ *blockDAG V* ∨ *a* ∉ *verts V* ∨ *b* ∉ *verts V* ∨ *c* ∉ *verts V*) *then 0 else*
if (*b*=*c*) *then 1 else*
if (((*a* →+ *V b*) ∨ *a* = *b*) ∧ ¬(*a* →+ *V c*)) *then 1 else*
if (((*a* →+ *V c*) ∨ *a* = *c*) ∧ ¬(*a* →+ *V b*)) *then* −*1 else*
if ((*a* →+ *V b*) ∧ (*a* →+ *V c*)) *then*
(*tie-break-int b c* (*signum* (*sum-list* (*map* (λ*i*.
(*vote-Spectre* (*reduce-past V a*) *i b c*)) (*sorted-list-of-set* (*past-nodes V a*))))))
else
signum (*sum-list* (*map* (λ*i*.

```
  (vote-Spectre V i b c)) (sorted-list-of-set (future-nodes V a)))))
  by auto
termination
proof
  let ?R = measures [( λ(V, a, b, c). (card (verts V))), ( λ(V, a, b, c). card {e.
e →*V a})]
  show wf ?R
    by simp
next
  fix V::('a::linorder, 'b) pre-digraph
  fix x a b c
  assume bD: ¬ (¬ blockDAG V ∨ a ∉ verts V ∨ b ∉ verts V ∨ c ∉ verts V)
  then have a ∈ verts V by simp
  then have card (verts (reduce-past V a)) < card (verts V)
    using bD blockDAG.reduce-less
    by metis
  then show ((reduce-past V a, x, b, c), V, a, b, c)
      ∈ measures
          [λ(V, a, b, c). card (verts V),
           λ(V, a, b, c). card {e. e →*V a}]
    by simp
next
  fix V::('a::linorder, 'b) pre-digraph
  fix x a b c
  assume bD: ¬ (¬ blockDAG V ∨ a ∉ verts V ∨ b ∉ verts V ∨ c ∉ verts V)
  then have a-in: a ∈ verts V using bD by simp
  assume x ∈ set (sorted-list-of-set (future-nodes V a))
  then have x ∈ future-nodes V a using DAG.finite-future
      set-sorted-list-of-set bD subs
    by metis
  then have rr: x →+V a using future-nodes.simps bD mem-Collect-eq
    by simp
  then have a-not: ¬ a →*V x using bD DAG.unidirectional subs by metis
  have bD2: blockDAG V using bD by simp
  have ∀ x. {e. e →*V x} ⊆ verts V using subs bD2 subsetI
      wf-digraph.reachable-in-verts(1) mem-Collect-eq
    by metis
  then have fin: ∀ x. finite {e. e →*V x} using subs bD2 fin-digraph.finite-verts
      finite-subset
    by metis
  have x →*V a using rr wf-digraph.reachable1-reachable subs bD2 by metis
  then have {e. e →*V x} ⊆ {e. e →*V a} using rr
      wf-digraph.reachable-trans Collect-mono subs bD2 by metis
  then have {e. e →*V x} ⊂ {e. e →*V a} using a-not
      subs bD2 a-in mem-Collect-eq psubsetI wf-digraph.reachable-refl
    by metis
  then have card {e. e →*V x} < card {e. e →*V a} using fin
    by (simp add: psubset-card-mono)
  then show ((V, x, b, c), V, a, b, c)
```

```
    ∈ measures
        [λ(V, a, b, c). card (verts V), λ(V, a, b, c). card {e. e →*_V a}]
  by simp
qed
```

Given vote-Spectre calculate if $a < b$ for arbitrary nodes

```
definition Spectre-Order :: ('a::linorder,'b) pre-digraph ⇒ 'a ⇒ 'a ⇒ bool
  where Spectre-Order G a b = ( tie-break-int a b (signum ( sum-list (map (λi.
   (vote-Spectre G i a b)) (sorted-list-of-set (verts G))))) = 1)
```

Given Spectre-Order calculate the corresponding relation over the nodes of G

```
definition SPECTRE :: ('a::linorder,'b) pre-digraph ⇒ ('a × 'a) set
  where SPECTRE G ≡ {(a,b) ∈ (verts G × verts G). Spectre-Order G a b}
```

4.2 Lemmas

```
lemma sumlist-one-mono:
  assumes ∀ x ∈ set L. x ≥ 0
    and ∃ x ∈ set L. x > 0
  shows signum (sum-list L) = 1
  using assms
proof(induct L, simp)
  case (Cons a2 L)
  consider (bg) a2 > 0 | a2 = 0 using Cons
    by (metis le-less list.set-intros(1))
  then show ?case
  proof(cases)
    case bg
    then have sum-list L ≥ 0  using Cons
      by (simp add: sum-list-nonneg)
    then have sum-list (a2 # L) > 0 using bg sum-list-def
      by auto
    then show ?thesis using tie-break-int.simps
      by auto
  next
    case 2
    then have  be: ∃ a∈set L. 0 < a using Cons
      by (metis less-int-code(1) set-ConsD)
    then have L ≠ [] by auto
    then show ?thesis using sum-list-def 2
      using Cons.hyps Cons.prems(1) be by auto
  qed
qed

lemma domain-signum: signum i ∈ {−1,0,1} by simp

lemma domain-tie-break:
  shows tie-break-int a b (signum i) ∈ {−1 ,1}
```

using *tie-break-int.simps*
by *auto*

lemma *Spectre-casesAlt*:
fixes V:: ('*a*::*linorder*,'*b*) *pre-digraph*
and a :: '*a*::*linorder* **and** b :: '*a*::*linorder* **and** c :: '*a*::*linorder*
obtains (*no-bD*) ($\neg$ *blockDAG* $V \vee a \notin$ *verts* $V \vee b \notin$ *verts* $V \vee c \notin$ *verts* V)
| (*equal*) (*blockDAG* $V \wedge a \in$ *verts* $V \wedge b \in$ *verts* $V \wedge c \in$ *verts* V) $\wedge$ $b = c$
| (*one*) (*blockDAG* $V \wedge a \in$ *verts* $V \wedge b \in$ *verts* $V \wedge c \in$ *verts* V) $\wedge$
$b \neq c \ \wedge (((a \rightarrow^+_V b) \vee a = b) \wedge \neg(a \rightarrow^+_V c))$
| (*two*) (*blockDAG* $V \wedge a \in$ *verts* $V \wedge b \in$ *verts* $V \wedge c \in$ *verts* V) $\wedge$ $b \neq c$
$\wedge \neg(((a \rightarrow^+_V b) \vee a = b) \wedge \neg(a \rightarrow^+_V c)) \wedge$
$((a \rightarrow^+_V c) \vee a = c) \wedge \neg(a \rightarrow^+_V b)$
| (*three*) (*blockDAG* $V \wedge a \in$ *verts* $V \wedge b \in$ *verts* $V \wedge c \in$ *verts* V) $\wedge$ $b \neq c$
$\wedge \neg(((a \rightarrow^+_V b) \vee a = b) \wedge \neg(a \rightarrow^+_V c)) \wedge$
$\neg(((a \rightarrow^+_V c) \vee a = c) \wedge \neg(a \rightarrow^+_V b)) \wedge$
$((a \rightarrow^+_V b) \wedge (a \rightarrow^+_V c))$
| (*four*) (*blockDAG* $V \wedge a \in$ *verts* $V \wedge b \in$ *verts* $V \wedge c \in$ *verts* V) $\wedge$ $b \neq c \ \wedge$
$\neg(((a \rightarrow^+_V b) \vee a = b) \wedge \neg(a \rightarrow^+_V c)) \wedge$
$\neg(((a \rightarrow^+_V c) \vee a = c) \wedge \neg(a \rightarrow^+_V b)) \wedge$
$\neg((a \rightarrow^+_V b) \wedge (a \rightarrow^+_V c))$
by *auto*

lemma *Spectre-theo*:
assumes *P 0*
and *P 1*
and *P* (*−1*)
and *P* (*tie-break-int b c* (*signum* (*sum-list* (*map* (*λi.*
(*vote-Spectre* (*reduce-past V a*) *i b c*)) (*sorted-list-of-set* ((*past-nodes V a*)))))))
and *P* (*signum* (*sum-list* (*map* (*λi.*
(*vote-Spectre V i b c*)) (*sorted-list-of-set* (*future-nodes V a*)))))
shows *P* (*vote-Spectre V a b c*)
using *assms vote-Spectre.simps*
by (*metis* (*mono-tags*, *lifting*))

lemma *domain-Spectre*:
shows *vote-Spectre V a b c* $\in \{-1, 0, 1\}$
proof(*rule Spectre-theo*, *auto*) **qed**

lemma *antisymmetric-tie-break*:
shows $b \neq c \implies$ *tie-break-int b c i* $= -$ *tie-break-int c b* $(-i)$
unfolding *tie-break-int.simps* **using** *less-not-sym* **by** *auto*

lemma *antisymmetric-sumlist*:
shows *sum-list* (*l*::*int list*) = − *sum-list* (*map* (λ*x*. −*x*) *l*)
proof(*induct l*, *auto*) **qed**

lemma *antisymmetric-signum*:
shows *signum i* = − (*signum* (−*i*))
by *auto*

lemma *vote-Spectre-one-exists*:
assumes *blockDAG V*
and *a* ∈ *verts V*
and *b* ∈ *verts V*
shows ∃ *i* ∈ *verts V*. *vote-Spectre V i a b* ≠ *0*
proof
show *a* ∈ *verts V* **using** *assms*(*2*) **by** *simp*
show *vote-Spectre V a a b* ≠ *0*
using *assms*
proof(*cases a b a V rule*: *Spectre-casesAlt*, *simp*, *simp*, *simp*, *simp*)
case *three*
then show *?thesis*
by (*meson DAG.cycle-free blockDAG.axioms*(*1*))
next
case *four*
then show *?thesis*
by *blast*
qed
qed

end

theory *Ghostdag*
imports *blockDAG Utils TopSort*
begin

5 GHOSTDAG

Based on the GHOSTDAG blockDAG consensus algorithmus by Sompolinsky and Zohar 2018

5.1 Funcitions and Definitions

Function to compare the size of set and break ties. Used for the GHOSTDAG maximum blue cluster selection

fun *larger-blue-tuple* ::
(('a::*linorder set* × 'a *list*) × 'a) ⇒ (('a *set* × 'a *list*) × 'a) ⇒ (('a *set* × 'a *list*) × 'a)
where *larger-blue-tuple A B* =
(*if* (*card* (*fst* (*fst A*))) > (*card* (*fst* (*fst B*))) ∨
(*card* (*fst* (*fst A*)) ≥ *card* (*fst* (*fst B*)) ∧ *snd A* ≤ *snd B*) *then A else B*)

Function to add node a to a tuple of a set S and List L

fun *add-set-list-tuple* :: (('a::*linorder set* × 'a *list*) × 'a) ⇒ ('a::*linorder set* × 'a *list*)
where *add-set-list-tuple* ((*S*,*L*),*a*) = (*S* ∪ {*a*}, *L* @ [*a*])

Function that adds a node a to a kCluster S, if $S + a$ remains a kCluster. Also adds a to the end of list L

fun *app-if-blue-else-add-end* ::
('a::*linorder*,'b) *pre-digraph* ⇒ *nat* ⇒ 'a ⇒ ('a::*linorder set* × 'a *list*)
⇒ ('a::*linorder set* × 'a *list*)
where *app-if-blue-else-add-end G k a* (*S*,*L*) = (*if* (*kCluster G k* (*S* ∪ {*a*}))
then add-set-list-tuple ((*S*,*L*),*a*) *else* (*S*,*L* @ [*a*]))

Function to select the largest $((S, L), a)$ according to $larger - blue - tuple$

fun *choose-max-blue-set* :: (('a::*linorder set* × 'a *list*) × 'a) *list* ⇒ (('a *set* × 'a *list*) × 'a)
where *choose-max-blue-set L* = *fold* (*larger-blue-tuple*) *L* (*hd L*)

GHOSTDAG ordering algorithm

function *OrderDAG* :: ('a::*linorder*,'b) *pre-digraph* ⇒ *nat* ⇒ ('a *set* × 'a *list*)
where
OrderDAG G k =
(*if* (¬ *blockDAG G*) *then* ({},[]) *else*
if (*card* (*verts G*) = *1*) *then* ({*genesis-nodeAlt G*},[*genesis-nodeAlt G*]) *else*
let M = *choose-max-blue-set*
((*map* (λ*i*.(((*OrderDAG* (*reduce-past G i*) *k*)) , *i*)) (*sorted-list-of-set* (*tips G*))))
in fold (*app-if-blue-else-add-end G k*) (*top-sort G* (*sorted-list-of-set* (*anticone G* (*snd M*))))
(*add-set-list-tuple M*))

by *auto*
termination proof
let *?R* = *measure* (λ(*G*, *k*). (*card* (*verts G*)))
show *wf ?R* **by** *auto*
next
fix *G*::('a::*linorder*,'b) *pre-digraph*
fix *k*::*nat*
fix *x*
assume *bD*: ¬ ¬ *blockDAG G*
assume *card* (*verts G*) ≠ *1*
then have *card* (*verts G*) > *1* **using** *bD blockDAG.blockDAG-size-cases* **by** *auto*

then have *nT*: $\forall x \in tips\ G.\ \neg\ blockDAG.is\text{-}genesis\text{-}node\ G\ x$
using *blockDAG.tips-unequal-gen bD tips-def mem-Collect-eq*
by *metis*
assume $x \in set\ (sorted\text{-}list\text{-}of\text{-}set\ (tips\ G))$
then have *in-t*: $x \in tips\ G$ **using** *bD*
by (*metis card-gt-0-iff length-pos-if-in-set length-sorted-list-of-set set-sorted-list-of-set*)

then show $((reduce\text{-}past\ G\ x,\ k),\ G,\ k) \in measure\ (\lambda(G,\ k).\ card\ (verts\ G))$
using *blockDAG.reduce-less bD tips-def is-tip.simps*
by *fastforce*
qed

Creating a relation on verts G based on the GHOSTDAG OrderDAG algorithm

fun $GHOSTDAG :: nat \Rightarrow ('a::linorder,'b)\ pre\text{-}digraph \Rightarrow 'a\ rel$
where $GHOSTDAG\ k\ G = list\text{-}to\text{-}rel\ (snd\ (OrderDAG\ G\ k))$

5.2 Soundness

lemma *OrderDAG-casesAlt*:
obtains (*ntB*) $\neg\ blockDAG\ G$
| (*one*) $blockDAG\ G \wedge card\ (verts\ G) = 1$
| (*more*) $blockDAG\ G \wedge card\ (verts\ G) > 1$
using *blockDAG.blockDAG-size-cases* **by** *auto*

5.2.1 Soundness of the $add - set - list$ function

lemma *add-set-list-tuple-mono*:
shows $set\ L \subseteq set\ (snd\ (add\text{-}set\text{-}list\text{-}tuple\ ((S,L),a)))$
using *add-set-list-tuple.simps* **by** *auto*

lemma *add-set-list-tuple-mono2*:
shows $set\ (snd\ (add\text{-}set\text{-}list\text{-}tuple\ ((S,L),a))) \subseteq set\ L \cup \{a\}$
using *add-set-list-tuple.simps* **by** *auto*

lemma *add-set-list-tuple-length*:
shows $length\ (snd\ (add\text{-}set\text{-}list\text{-}tuple\ ((S,L),a))) = Suc\ (length\ L)$
proof(*induct L*, *auto*) **qed**

5.2.2 Soundness of the $add - if - blue$ function

lemma *app-if-blue-mono*:
assumes *finite S*
shows $(fst\ (S,L)) \subseteq (fst\ (app\text{-}if\text{-}blue\text{-}else\text{-}add\text{-}end\ G\ k\ a\ (S,L)))$
unfolding *app-if-blue-else-add-end.simps add-set-list-tuple.simps*
by (*simp add*: *assms card-mono subset-insertI*)

lemma *app-if-blue-mono2*:
shows $set\ (snd\ (S,L)) \subseteq set\ (snd\ (app\text{-}if\text{-}blue\text{-}else\text{-}add\text{-}end\ G\ k\ a\ (S,L)\ ))$

unfolding *app-if-blue-else-add-end.simps add-set-list-tuple.simps*
by (*simp add*: *subsetI*)

lemma *app-if-blue-append*:
shows $a \in set\ (snd\ (app\text{-}if\text{-}blue\text{-}else\text{-}add\text{-}end\ G\ k\ a\ (S,L))\ )$
unfolding *app-if-blue-else-add-end.simps add-set-list-tuple.simps*
by *simp*

lemma *app-if-blue-mono3*:
shows $set\ (snd\ (app\text{-}if\text{-}blue\text{-}else\text{-}add\text{-}end\ G\ k\ a\ (S,L))) \subseteq set\ L \cup \{a\}$
unfolding *app-if-blue-else-add-end.simps add-set-list-tuple.simps*
by (*simp add*: *subsetI*)

lemma *app-if-blue-mono4*:
assumes $set\ L1 \subseteq set\ L2$
shows $set\ (snd\ (app\text{-}if\text{-}blue\text{-}else\text{-}add\text{-}end\ G\ k\ a\ (S,L1)))$
$\subseteq set\ (snd\ (app\text{-}if\text{-}blue\text{-}else\text{-}add\text{-}end\ G\ k\ a\ (S2,L2)))$
unfolding *app-if-blue-else-add-end.simps add-set-list-tuple.simps*
using *assms* **by** *auto*

lemma *app-if-blue-card-mono*:
assumes *finite S*
shows $card\ (fst\ (S,L)) \leq card\ (fst\ (app\text{-}if\text{-}blue\text{-}else\text{-}add\text{-}end\ G\ k\ a\ (S,L)))$
unfolding *app-if-blue-else-add-end.simps add-set-list-tuple.simps*
by (*simp add*: *assms card-mono subset-insertI*)

lemma *app-if-blue-else-add-end-length*:
shows $length\ (snd\ (app\text{-}if\text{-}blue\text{-}else\text{-}add\text{-}end\ G\ k\ a\ (S,L))) = Suc\ (length\ \ L)$
proof(*induction L*, *auto*) **qed**

5.2.3 Soundness of the $larger - blue - tuple$ comparison

lemma *larger-blue-tuple-mono*:
assumes *finite* (*fst V*)
shows *larger-blue-tuple* ((*app-if-blue-else-add-end G k a V*),*b*) (*V*,*b*)
$=$ ((*app-if-blue-else-add-end G k a V*),*b*)
using *assms app-if-blue-card-mono larger-blue-tuple.simps eq-refl*
by (*metis fst-conv prod.collapse snd-conv*)

lemma *larger-blue-tuple-subs*:
shows *larger-blue-tuple A B* $\in \{A,B\}$ **by** *auto*

5.2.4 Soundness of the $choose_max_blue_set$ function

lemma *choose-max-blue-avoid-empty*:

assumes $L \neq []$
shows *choose-max-blue-set L* $\in$ *set L*
unfolding *choose-max-blue-set.simps*
proof (*rule fold-invariant*)
show $\bigwedge x.\ x \in set\ L \Longrightarrow x \in set\ L$ **using** *assms* **by** *auto*
next
show *hd L* $\in$ *set L* **using** *assms* **by** *auto*
next
fix *x s*
assume *x* $\in$ *set L*
and *s* $\in$ *set L*
then show *larger-blue-tuple x s* $\in$ *set L* **using** *larger-blue-tuple.simps* **by** *auto*
qed

5.2.5 Auxiliary lemmas for OrderDAG

lemma *fold-app-length*:
shows *length* (*snd* (*fold* (*app-if-blue-else-add-end G k*)
L1 PL2)) = *length L1* + *length* (*snd PL2*)
proof(*induct L1 arbitrary*: *PL2*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a L1*)
then show *?case* **unfolding** *fold-Cons comp-apply* **using** *app-if-blue-else-add-end-length*
by (*metis add-Suc add-Suc-right length-Cons old.prod.exhaust snd-conv*)
qed

lemma *fold-app-mono*:
shows *snd* (*fold* (*app-if-blue-else-add-end G k*) *L2* (*S*,*L1*)) = *L1* @ *L2*
proof(*induct L2 arbitrary*: *S L1*, *simp*)
case (*Cons a L2*)
then show *?case* **unfolding** *fold-simps*(*2*) **using** *app-if-blue-else-add-end.simps*
by *simp*
qed

lemma *fold-app-mono1*:
assumes *x* $\in$ *set* (*snd* (*S*,*L1*))
shows *x* $\in$ *set* (*snd* (*fold* (*app-if-blue-else-add-end G k*) *L2* (*S2*,*L1*)))
using *fold-app-mono*
by (*metis Cons-eq-appendI append.assoc assms in-set-conv-decomp sndI*)

lemma *fold-app-mono2*:
assumes *x* $\in$ *set L2*
shows *x* $\in$ *set* (*snd* (*fold* (*app-if-blue-else-add-end G k*) *L2* (*S*,*L1*)))
using *assms* **unfolding** *fold-app-mono* **by** *auto*

lemma *fold-app-mono3*:

assumes *set L1* $\subseteq$ *set L2*
shows *set* (*snd* (*fold* (*app-if-blue-else-add-end G k*) *L* (*S1*, *L1*)))
$\subseteq$ *set* (*snd* (*fold* (*app-if-blue-else-add-end G k*) *L* (*S2*, *L2*)))
using *assms* **unfolding** *fold-app-mono*
by *auto*

lemma *fold-app-mono-ex*:
shows *set* (*snd* (*fold* (*app-if-blue-else-add-end G k*) *L2* (*S*,*L1*))) = (*set L2* $\cup$ *set L1*)
unfolding *fold-app-mono* **by** *auto*

lemma *fold-app-mono-rel*:
assumes (*x*,*y*) $\in$ *list-to-rel L1*
shows (*x*,*y*) $\in$ *list-to-rel* (*snd* (*fold* (*app-if-blue-else-add-end G k*) *L2* (*S*,*L1*)))
using *assms*
proof(*induct L2 arbitrary*: *S L1*, *simp*)
case (*Cons a L2*)
then show *?case*
unfolding *fold.simps*(*2*) *comp-apply*
using *list-to-rel-mono app-if-blue-else-add-end.simps*
by (*metis add-set-list-tuple.simps prod.collapse snd-conv*)
qed

lemma *fold-app-mono-rel2*:
assumes (*x*,*y*) $\in$ *list-to-rel L2*
shows (*x*,*y*) $\in$ *list-to-rel* (*snd* (*fold* (*app-if-blue-else-add-end G k*) *L2* (*S*,*L1*)))
using *assms*
by (*simp add*: *fold-app-mono list-to-rel-mono2*)

lemma *fold-app-app-rel*:
assumes *x* $\in$ *set L1*
and *y* $\in$ *set L2*
shows (*x*,*y*) $\in$ *list-to-rel* (*snd* (*fold* (*app-if-blue-else-add-end G k*) *L2* (*S*,*L1*)))
using *assms*
proof(*induct L2 arbitrary*: *S L1*, *simp*)
case (*Cons a L2*)
then show *?case*
unfolding *fold.simps*(*2*) *comp-apply*
using *list-to-rel-append app-if-blue-else-add-end.simps*
by (*metis Un-iff add-set-list-tuple.simps fold-app-mono-rel set-ConsD set-append*)

qed

lemma *chosen-max-tip*:
assumes *blockDAG G*
assumes *x* = *snd* (*choose-max-blue-set* (*map* (λi. (*OrderDAG* (*reduce-past G i*) *k*, *i*))

```
      (sorted-list-of-set (tips G))))
  shows  x ∈ set (sorted-list-of-set (tips G)) and  x ∈ tips G
proof −
  obtain pp where pp-in: pp =  (map (λi. (OrderDAG (reduce-past G i) k, i))
   (sorted-list-of-set (tips G))) using blockDAG.tips-exist by auto
 have mm: choose-max-blue-set pp ∈ set pp using pp-in choose-max-blue-avoid-empty
      digraph.tips-finite subs assms(1)
      list.map-disc-iff sorted-list-of-set-eq-Nil-iff blockDAG.tips-not-empty
    by (metis (mono-tags, lifting))
  then have kk: snd (choose-max-blue-set pp) ∈ set (map  snd pp)
    by auto
  have mm2: ⋀L. (map snd (map (λi. ((OrderDAG (reduce-past G i) k) , i)) L))
=  L
  proof −
    fix L
    show map snd (map (λi. (OrderDAG (reduce-past G i) k, i)) L) = L
    proof(induct L)
      case Nil
      then show ?case by auto
    next
      case (Cons a L)
      then show ?case by auto
    qed
  qed
  have set (map snd pp) = set (sorted-list-of-set (tips G))
    using mm2 pp-in  by auto
  then show x ∈ set (sorted-list-of-set (tips G)) using pp-in assms(2) kk by blast

  then show x ∈ tips G
    using digraph.tips-finite sorted-list-of-set(1) kk subs assms pp-in by auto
qed


lemma chosen-map-simps1:
  assumes  x ∈ set  (map (λi. (P i, i)) L)
  shows  fst x = P (snd x)
  using assms
proof(induct L, auto) qed

lemma chosen-map-simps:
  assumes blockDAG G
  assumes x = map (λi. (OrderDAG (reduce-past G i) k, i))
      (sorted-list-of-set (tips G))
  shows  snd  (choose-max-blue-set x) ∈ set (sorted-list-of-set (tips G))
    and  snd (choose-max-blue-set x) ∈ tips G
    and set (map snd x) = set (sorted-list-of-set (tips G))
    and choose-max-blue-set x ∈ set x
    and ¬ blockDAG.is-genesis-node G (snd (choose-max-blue-set x)) ⟹
  blockDAG (reduce-past G (snd (choose-max-blue-set x)))
```

and *OrderDAG* (*reduce-past G* (*snd* (*choose-max-blue-set x*))) *k* = *fst* (*choose-max-blue-set x*)
proof −
obtain *pp* **where** *pp-in*: *pp* = (*map* (λ*i*. (*OrderDAG* (*reduce-past G i*) *k*, *i*)) (*sorted-list-of-set* (*tips G*))) **using** *blockDAG.tips-exist* **by** *auto*
have *mm*: *choose-max-blue-set pp* ∈ *set pp* **using** *pp-in choose-max-blue-avoid-empty digraph.tips-finite subs assms*(*1*) *list.map-disc-iff sorted-list-of-set-eq-Nil-iff blockDAG.tips-not-empty* **by** (*metis* (*mono-tags*, *lifting*))
then have *kk*: *snd* (*choose-max-blue-set pp*) ∈ *set* (*map snd pp*) **by** *auto*
have *seteq*: *set* (*map snd pp*) = *set* (*sorted-list-of-set* (*tips G*)) **using** *map-snd-map pp-in* **by** *auto*
then show *snd* (*choose-max-blue-set x*) ∈ *set* (*sorted-list-of-set* (*tips G*)) **using** *pp-in assms*(*2*) *kk* **by** *blast*
then show *tip*: *snd* (*choose-max-blue-set x*) ∈ *tips G* **using** *digraph.tips-finite sorted-list-of-set*(*1*) *kk subs assms pp-in* **by** *auto*
show *set* (*map snd x*) = *set* (*sorted-list-of-set* (*tips G*)) **using** *map-snd-map assms*(*2*) **by** *simp*
then show *choose-max-blue-set x* ∈ *set x* **using** *seteq pp-in assms*(*2*) *mm* **by** *blast*
show *OrderDAG* (*reduce-past G* (*snd* (*choose-max-blue-set x*))) *k* = *fst* (*choose-max-blue-set x*) **by** (*metis* (*no-types*) *assms*(*2*) *chosen-map-simps1 mm pp-in*)
assume ¬ *blockDAG.is-genesis-node G* (*snd* (*choose-max-blue-set x*))
then show *blockDAG* (*reduce-past G* (*snd* (*choose-max-blue-set x*))) **using** *tip blockDAG.reduce-past-dagbased assms*(*1*) *digraph.tips-in-verts subs subsetD* **by** *metis*
qed

5.2.6 OrderDAG soundness

lemma *Verts-in-OrderDAG*:
assumes *blockDAG G* **and** *x* ∈ *verts G*
shows *x* ∈ *set* (*snd* (*OrderDAG G k*))
using *assms*
proof(*induct G k arbitrary*: *x rule*: *OrderDAG.induct*)
case (*1 G k x*)
then have *bD*: *blockDAG G* **by** *auto*
assume *x-in*: *x* ∈ *verts G*
then consider (*cD1*) *card* (*verts G*) = *1*| (*cDm*) *card* (*verts G*) ≠ *1* **by** *auto*
then show *x* ∈ *set* (*snd* (*OrderDAG G k*))
proof(*cases*)
case (*cD1*)
then have *set* (*snd* (*OrderDAG G k*)) = {*genesis-nodeAlt G*} **using** *1 OrderDAG.simps* **by** *auto*

```
    then show ?thesis using x-in bD cD1
        genesis-nodeAlt-sound blockDAG.is-genesis-node.simps
      using 1
      by (metis card-1-singletonE singletonD)
  next
    case (cDm)
    then show ?thesis
    proof −
      obtain pp where pp-in: pp =  (map (λi. (OrderDAG (reduce-past G i) k, i))
       (sorted-list-of-set (tips G))) using blockDAG.tips-exist by auto
      then have tt2: snd (choose-max-blue-set pp) ∈ tips G
        using chosen-map-simps bD
        by blast
      show ?thesis
      proof(rule blockDAG.tips-cases)
        show blockDAG G using bD by auto
        show snd (choose-max-blue-set pp) ∈ tips G using tt2 by auto
        show x ∈ verts G using x-in by auto
      next
        assume as1: x = snd (choose-max-blue-set pp)
        obtain fCur where fcur-in: fCur = add-set-list-tuple (choose-max-blue-set
pp)
          by auto
        have x ∈ set (snd(fCur))
          unfolding as1 using  add-set-list-tuple.simps fcur-in
            add-set-list-tuple.cases snd-conv insertI1 snd-conv
           by (metis (mono-tags, hide-lams) Un-insert-right fst-conv list.simps(15)
set-append)
        then have x ∈ set (snd (fold (app-if-blue-else-add-end G k)
                (top-sort G (sorted-list-of-set (anticone G (snd (choose-max-blue-set
pp))))) (fCur)))
          using  fold-app-mono1 surj-pair
          by (metis)
       then show ?thesis unfolding pp-in fcur-in using 1 OrderDAG.simps cDm
          by (metis (mono-tags, lifting))
      next
        assume anti: x ∈ anticone G (snd (choose-max-blue-set pp))
       obtain ttt where ttt-in: ttt = add-set-list-tuple (choose-max-blue-set pp) by
auto
        have x ∈ set (snd (fold (app-if-blue-else-add-end G k)
              (top-sort G (sorted-list-of-set (anticone G (snd (choose-max-blue-set
pp)))))
                 ttt))
          using pp-in sorted-list-of-set(1) anti bD subs
            DAG.anticon-finite fold-app-mono2 surj-pair top-sort-con  by metis
        then show x ∈ set (snd (OrderDAG G k)) using OrderDAG.simps pp-in
bD cDm ttt-in 1
          by (metis (no-types, lifting) map-eq-conv)
      next
```

```
      assume  as2: x ∈ past-nodes G (snd (choose-max-blue-set pp))
      then have pas: x ∈ verts (reduce-past G (snd (choose-max-blue-set pp)))
        using reduce-past.simps induce-subgraph-verts by auto
      have cd1: card (verts G) > 1 using cDm bD
        using blockDAG.blockDAG-size-cases by blast
     have (snd (choose-max-blue-set pp)) ∈ set (sorted-list-of-set (tips G)) using
tt2
          digraph.tips-finite bD subs sorted-list-of-set(1) by auto
      moreover
      have blockDAG (reduce-past G (snd (choose-max-blue-set pp))) using
          blockDAG.reduce-past-dagbased bD tt2  blockDAG.tips-unequal-gen
          cd1 tips-def CollectD by metis
      ultimately have bass:
        x ∈ set ((snd (OrderDAG (reduce-past G (snd (choose-max-blue-set pp)))
k)))
        using  pp-in 1 cDm tt2 pas by metis
      then have in-F: x ∈ set (snd ( fst ((choose-max-blue-set pp))))
        using x-in chosen-map-simps(6) pp-in
        using bD by fastforce
      then have x ∈ set (snd (fold (app-if-blue-else-add-end G k)
      (top-sort G (sorted-list-of-set (anticone G (snd (choose-max-blue-set pp)))))
       (fst((choose-max-blue-set pp)))))
        by (metis fold-app-mono1 in-F prod.collapse)
      moreover have OrderDAG G k = (fold (app-if-blue-else-add-end G k)
      (top-sort G (sorted-list-of-set (anticone G (snd (choose-max-blue-set pp))))))
     (add-set-list-tuple (choose-max-blue-set pp))) using cDm 1 OrderDAG.simps
pp-in
        by (metis (no-types, lifting) map-eq-conv)
      then show x ∈ set (snd (OrderDAG G k))
        by (metis (no-types, lifting) add-set-list-tuple-mono fold-app-mono1
          in-F prod.collapse subset-code(1))
    qed
   qed
  qed
qed


lemma OrderDAG-in-verts:
  assumes x ∈ set (snd (OrderDAG G k))
  shows x ∈ verts G
  using assms
proof(induction G k arbitrary: x rule: OrderDAG.induct)
  case (1 G k x)
  consider (inval) ¬ blockDAG G| (one) blockDAG G ∧
  card (verts G) = 1 | (val) blockDAG G ∧
  card (verts G) ≠ 1 by auto
  then show ?case
  proof(cases)
    case inval
```

```
  then show ?thesis using 1 by auto
 next
  case one
 then show ?thesis using OrderDAG.simps 1 genesis-nodeAlt-one-sound blockDAG.is-genesis-node.simps
    using empty-set list.simps(15) singleton-iff sndI by fastforce
 next
  case val
  then show ?thesis
  proof
    have bD: blockDAG G using val  by auto
      obtain M where M-in:M = choose-max-blue-set (map (λi. (OrderDAG
(reduce-past G i) k, i))
     (sorted-list-of-set (tips G))) by auto
    obtain pp where pp-in: pp =  (map (λi. (OrderDAG (reduce-past G i) k, i))
     (sorted-list-of-set (tips G))) using blockDAG.tips-exist by auto
    have set (snd (OrderDAG G k)) =
      set (snd (fold (app-if-blue-else-add-end G k) (top-sort G (sorted-list-of-set
(anticone G (snd M))))
    (add-set-list-tuple M))) unfolding M-in val using OrderDAG.simps val
      by (metis (mono-tags, lifting))
    then have set (snd (OrderDAG G k))
  = set (top-sort G (sorted-list-of-set (anticone G (snd M)))) ∪ set (snd (add-set-list-tuple
M))
      using fold-app-mono-ex
      by (metis eq-snd-iff)
     then consider (ac) x ∈ set (top-sort G (sorted-list-of-set (anticone G (snd
M))))
      | (co) x ∈ set (snd (add-set-list-tuple M))
      using 1 by auto
    then show x ∈ verts G proof(cases)
      case ac
      then show ?thesis using top-sort-con DAG.anticone-in-verts val
          sorted-list-of-set(1) subs
        by (metis DAG.anticon-finite subsetD)
    next
      case co
      then consider (ma) x = snd M | (nma) x ∈ set (snd( fst(M)))
        using add-set-list-tuple.simps
        by (metis (no-types, lifting) Un-insert-right append-Nil2 insertE
            list.simps(15) prod.collapse set-append sndI)
      then show ?thesis proof(cases)
        case ma
        then show ?thesis unfolding M-in  using bD
            chosen-map-simps(2) digraph.tips-in-verts subs
          by blast
      next
        have mm: choose-max-blue-set pp ∈ set pp unfolding pp-in using bD
chosen-map-simps(4)
      by (metis (mono-tags, lifting) Nil-is-map-conv choose-max-blue-avoid-empty)
```

```
        case nma
        then have x ∈ set (snd (OrderDAG (reduce-past G (snd M)) k))
          unfolding M-in choose-max-blue-avoid-empty blockDAG.tips-not-empty
bD
          by (metis (no-types, lifting) ex-map-conv fst-conv mm pp-in snd-conv)
       then have x ∈ verts (reduce-past G (snd M)) using 1 val chosen-map-simps
M-in pp-in
            sorted-list-of-set(1) digraph.tips-finite subs bD
          by blast
          then show x ∈ verts G using reduce-past.simps induce-subgraph-verts
past-nodes.simps
          by auto
       qed
     qed
   qed
  qed
qed


lemma OrderDAG-length:
  shows blockDAG G ⟹ length (snd (OrderDAG G k)) = card (verts G)
proof(induct G k rule: OrderDAG.induct)
  case (1 G k)
  then show ?case proof (cases G rule: OrderDAG-casesAlt)
    case ntB
    then show ?thesis using 1 by auto
  next
    case one
    then show ?thesis using OrderDAG.simps by auto
  next
    case more
    show ?thesis using 1
    proof −
      have bD: blockDAG G using 1 by auto
      obtain ma where pp-in: ma = (choose-max-blue-set (map (λi. (OrderDAG
(reduce-past G i) k, i))
       (sorted-list-of-set (tips G))))
        by (metis)
      then have backw: OrderDAG G k = fold (app-if-blue-else-add-end G k)
            (top-sort G (sorted-list-of-set (anticone G (snd ma))))
            (add-set-list-tuple ma) using OrderDAG.simps pp-in more
        by (metis (mono-tags, lifting) less-numeral-extra(4))
     have tt: snd ma ∈ set (sorted-list-of-set (tips G)) using pp-in chosen-max-tip

        more by auto
      have ttt: snd ma ∈ tips G using chosen-max-tip(2) pp-in
        more by auto
    then have bD2: blockDAG (reduce-past G (snd ma)) using blockDAG.tips-unequal-gen
```

bD more
blockDAG.reduce-past-dagbased bD tips-def
by *fastforce*
then have *length* (*snd* (*OrderDAG* (*reduce-past G* (*snd ma*)) *k*))
= *card* (*verts* (*reduce-past G* (*snd ma*)))
using *1 tt bD2 more* **by** *auto*
then have *length* (*snd* (*fst ma*))
= *card* (*verts* (*reduce-past G* (*snd ma*)))
using *bD chosen-map-simps*(*6*) *pp-in*
by *fastforce*
then have *length* (*snd* (*add-set-list-tuple ma*)) = *1* + *card* (*verts* (*reduce-past G* (*snd ma*)))
by (*metis add-set-list-tuple-length plus-1-eq-Suc prod.collapse*)
then show *?thesis* **unfolding** *backw*
using *subs DAG.verts-size-comp ttt*
add.assoc add.commute bD fold-app-length length-sorted-list-of-set top-sort-len
by (*metis* (*full-types*))
qed
qed
qed

lemma *OrderDAG-total*:
assumes *blockDAG G*
shows *set* (*snd* (*OrderDAG G k*)) = *verts G*
using *Verts-in-OrderDAG OrderDAG-in-verts assms*(*1*)
by *blast*

lemma *OrderDAG-distinct*:
assumes *blockDAG G*
shows *distinct* (*snd* (*OrderDAG G k*))
using *OrderDAG-length OrderDAG-total*
card-distinct assms
by *metis*

end
theory *ExtendblockDAG*
imports *blockDAG*
begin

6 Extend blockDAGs

6.1 Definitions

locale *Append-One* = *blockDAG* +

fixes *G-A*::('*a*,'*b*) *pre-digraph* (**structure**)
and *app*::'*a*
assumes *bD-A*: *blockDAG G-A*
and *app-in*: *app* ∈ *verts G-A*
and *app-notin*: *app* ∉ *verts G*
and *GG-A* : *G* = *pre-digraph.del-vert G-A app*
and *new-node*: ∀ *b* ∈ *verts G-A*. ¬ *b* →$_{G\text{-}A}$ *app*

locale *Honest-Append-One* = *Append-One* +
assumes *ref-tips*: ∀ *t* ∈ *tips G*. *app* →$_{G\text{-}A}$ *t*

locale *Append-One-Honest-Dishonest* = *Honest-Append-One* +
fixes *G-AB* (**structure**)
and *dis-n*::'*a*
assumes *Append-One G-A G-AB dis-n*

6.2 Append-One Lemmas

lemma (**in** *Append-One*) *new-node-alt*:
(∀ *b* ∈ *verts G-A*. ¬ *b* →$_{G\text{-}A}$ *app*) ⟷ (∀ *b*. ¬ *b* →$_{G\text{-}A}$ *app*)
proof(*auto*)
fix *b*
assume *a1*: ∀ *b*∈*verts G-A*. (*b*, *app*) ∉ *arcs-ends G-A*
and *a2*: *b* →$_{G\text{-}A}$ *app*
then have *b* ∈ *verts G-A* **using** *wf-digraph.adj-in-verts*(*1*) *bD-A subs* **by** *metis*
then show *False* **using** *a1 a2* **by** *auto*
qed

lemma (**in** *Append-One*) *append-subverts*:
verts G ⊂ *verts G-A*
unfolding *GG-A pre-digraph.verts-del-vert* **using** *app-in app-notin* **by** *auto*

lemma (**in** *Append-One*) *append-verts*:
verts G-A = *verts G* ∪ {*app*}
unfolding *GG-A pre-digraph.verts-del-vert* **using** *app-in app-notin* **by** *auto*

lemma (**in** *Append-One*) *append-verts-in*:
assumes *a* ∈ *verts G*
shows *a* ∈ *verts G-A*
unfolding *append-verts*
by (*simp add*: *assms*)

lemma (**in** *Append-One*) *append-verts-diff*:
shows *verts G* = *verts G-A* − {*app*}
using *append-verts app-in app-notin* **by** *auto*

lemma (**in** *Append-One*) *append-verts-cases*:
assumes *a* ∈ *verts G-A*

obtains (*a-in-G*) $a \in verts\ G$ | (*a-eq-app*) $a = app$
using *append-verts assms* **by** *auto*

lemma (**in** *Append-One*) *append-subarcs-leq*:
$arcs\ G \subseteq arcs\ G\text{-}A$
unfolding *GG-A pre-digraph.arcs-del-vert* **using** *app-in app-notin*
using *wf-digraph-def subs Append-One-axioms* **by** *blast*

lemma (**in** *Append-One*) *append-subarcs*:
$arcs\ G \subset arcs\ G\text{-}A$
proof
show $arcs\ G \subseteq arcs\ G\text{-}A$ **using** *append-subarcs-leq* **by** *simp*
obtain *gen* **where** *gen-rec*: $app\ \rightarrow^+_{G\text{-}A}\ gen$ **using** *bD-A blockDAG.genesis app-in*
app-notin append-subverts
genesis-in-verts new-node psubsetD tranclE new-node-alt
by (*metis* (*mono-tags*, *lifting*))
then obtain *walk* **where** *walk-in*: *pre-digraph.awalk G-A app walk gen* $\wedge$ *walk* $\neq []$
using *wf-digraph.reachable1-awalk bD-A subs*
by *metis*
then obtain *e* **where** $\exists es.\ walk = e\ \#\ es$
by (*metis list.exhaust*)
then have *e-in*: $e \in arcs\ G\text{-}A \wedge tail\ G\text{-}A\ e = app$
using *wf-digraph.awalk-simps*(*2*)
bD-A subs walk-in
by *metis*
then have $e \notin arcs\ G$ **using** *wf-digraph-def app-notin*
blockDAG-axioms subs GG-A pre-digraph.tail-del-vert
by *metis*
then show $arcs\ G \neq arcs\ G\text{-}A$ **using** *e-in* **by** *auto*
qed

lemma (**in** *Append-One*) *append-head*:
$head\ G\text{-}A = head\ G$
using *GG-A*
by (*simp add*: *pre-digraph.head-del-vert*)

lemma (**in** *Append-One*) *append-tail*:
$tail\ G\text{-}A = tail\ G$
using *GG-A*
by (*simp add*: *pre-digraph.tail-del-vert*)

lemma (**in** *Append-One*) *append-subgraph*:
subgraph G G-A
using *GG-A blockDAG-axioms subs bD-A*
by (*simp add*: *subs wf-digraph.subgraph-del-vert*)

lemma (**in** *Append-One*) *append-induced-subgraph*:

induced-subgraph G G-A
unfolding *induced-subgraph-def*
proof
show *subgraph G G-A* **using** *append-subgraph* **by** *simp*
show *arcs G* = {*e* ∈ *arcs G-A*. *tail G-A e* ∈ *verts G* ∧ *head G-A e* ∈ *verts G*}
unfolding *GG-A pre-digraph.arcs-del-vert pre-digraph.verts-del-vert*
using *append-verts bD-A subs wf-digraph-def*
by (*metis* (*no-types*, *lifting*) *Diff-insert-absorb Un-empty-right*
Un-insert-right app-notin insertE)
qed

lemma (**in** *Append-One*) *append-not-reached*:
∀ *b* ∈ *verts G-A*. ¬ *b* →$^+_{G\text{-}A}$ *app*
using *tranclE wf-digraph.reachable1-in-verts*(*2*) *bD-A subs new-node*
by *metis*

lemma (**in** *Append-One*) *append-is-tip*:
is-tip G-A app
unfolding *is-tip.simps*
using *app-in new-node append-not-reached*
by *metis*

lemma (**in** *Append-One*) *append-in-tips*:
app ∈ *tips G-A*
unfolding *tips-def*
using *app-in new-node append-is-tip CollectI*
by *metis*

lemma (**in** *Append-One*) *append-greater-1*:
card (*verts G-A*) > *1*
unfolding *append-verts*
using *app-notin no-empty-blockDAG* **by** *auto*

lemma *append-diff-sorted-set*:
assumes *a* ∈ *A*
and *finite A*
shows *sum-list* ((*map* (*P*::('*a*::*linorder* ⇒ *int*)))
(*sorted-list-of-set* (*A* − {*a*})))
= *sum-list* ((*map P*)(*sorted-list-of-set* (*A*))) − (*P a*)
proof −
let *?L1* = (*sorted-list-of-set* (*A*))
have *d-1*: *distinct ?L1* **using** *sum-list-distinct-conv-sum-set sorted-list-of-set*(*2*)
by *auto*
then have *s-1*: *sum-list* ((*map P*) *?L1*)
= *sum P* (*set ?L1*) **using** *sum-list-distinct-conv-sum-set* **by** *metis*
let *?L2* = (*sorted-list-of-set* (*A* − {*a*}))
have *d-2*: *distinct ?L2* **using** *sum-list-distinct-conv-sum-set sorted-list-of-set*(*2*)

by *auto*
then have *s-2*: *sum-list* ((*map P*) *?L2*)
= *sum P* (*set ?L2*) **using** *sum-list-distinct-conv-sum-set* **by** *metis*
have *s-3*: *sum P* (*set ?L2*) = *sum P* (*set ?L1*) − (*P a*)
using *assms sorted-list-of-set*(*1*)
by (*simp add*: *sum-diff1*)
show *?thesis*
unfolding *s-1 s-2 s-3* **by** *simp*
qed

6.3 Honest-Append-One Lemmas

lemma (**in** *Honest-Append-One*) *reaches-all*:
∀ *v* ∈ *verts G*. *app* $\rightarrow^+_{G\text{-}A}$ *v*
proof
fix *v*
assume *v-in*: *v* ∈ *verts G*
consider *is-tip G v* | ¬ *is-tip G v* **by** *auto*
then show *app* $\rightarrow^+_{G\text{-}A}$ *v*
proof(*cases*)
case *1*
then show *?thesis* **using** *ref-tips v-in is-tip.simps r-into-trancl′ reached-by*
by (*metis*)
next
case *2*
then have *v* ∉ *tips G* **unfolding** *tips-def* **by** *simp*
then obtain *t* **where** *t-in*: *t* ∈ *tips G* ∧ *t* $\rightarrow^+_{G}$ *v*
using *reached-by-tip v-in* **by** *auto*
then have *t* $\rightarrow^+_{G\text{-}A}$ *v* **using** *v-in append-subgraph arcs-ends-mono trancl-mono*
by (*metis*)
moreover have *app* $\rightarrow^+_{G\text{-}A}$ *t*
using *ref-tips v-in r-into-trancl′ t-in*
by (*metis*)
ultimately show *?thesis* **using** *trancl-trans* **by** *auto*
qed
qed

lemma (**in** *Honest-Append-One*) *append-past-all*:
past-nodes G-A app = *verts G*
unfolding *past-nodes.simps append-verts*
using *reaches-all DAG.cycle-free bD-A subs*
by *fastforce*

lemma (**in** *Honest-Append-One*) *append-is-only-tip*:
tips G-A = {*app*}
proof *safe*
show *app* ∈ *tips G-A* **using** *append-in-tips* **by** *simp*
fix *x*
assume *as1*: *x* ∈ *tips G-A*

```
  then have x-in: x ∈ verts G-A using digraph.tips-in-verts bD-A subs by auto
  show x = app
  proof(rule ccontr)
    assume x ≠ app
    then have x ∈ verts G using append-verts x-in by auto
    then have app →⁺ G-A x using reaches-all by auto
    then have ¬ is-tip G-A x  unfolding is-tip.simps using app-in by auto
    then show False using as1 CollectD unfolding tips-def by auto
  qed
qed

lemma (in Honest-Append-One) reduce-append:
  reduce-past G-A app = G
  unfolding reduce-past.simps past-nodes.simps
proof −
  have {b ∈ verts G. app →⁺ G-A b} = verts G
    using reaches-all by auto
  moreover have {b ∈ verts G. app →⁺ G-A b} = {b ∈ verts G-A. app →⁺ G-A
b}
    unfolding append-verts using append-is-tip by fastforce
  ultimately have {b ∈ verts G-A. app →⁺ G-A b} = verts G by simp
  then show G-A ↾ {b ∈ verts G-A. app →⁺ G-A b} = G
    unfolding induce-subgraph-def
    using append-induced-subgraph induced-subgraph-def
    append-head append-tail
    by (metis (no-types, lifting) Collect-cong app-notin arcs-del-vert
        del-vert-def del-vert-not-in-graph verts-del-vert)
qed

lemma (in Honest-Append-One) append-no-anticone:
  anticone G-A app = {}
  unfolding anticone.simps
proof safe
  fix x
  assume  x ∈ verts G-A
  and  app ≠ x
  and as: (app, x) ∉ (arcs-ends G-A)⁺
  then have x ∈ verts G
    using append-verts by auto
  then have app →⁺ G-A x
    using reaches-all by auto
  then show x →⁺ G-A app
    using as by auto
qed
end
theory Properties
  imports blockDAG ExtendblockDAG Spectre Ghostdag
begin
```

definition *Linear-Order*:: $(('a,'b)\ pre\text{-}digraph \Rightarrow 'a\ rel) \Rightarrow bool$
where $Linear\text{-}Order\ A \equiv (\forall\, G.\ blockDAG\ G \longrightarrow linear\text{-}order\text{-}on\ (verts\ G)\ (A\ G))$

definition *Order-Preserving*:: $(('a,'b)\ pre\text{-}digraph \Rightarrow 'a\ rel) \Rightarrow bool$
where $Order\text{-}Preserving\ A \equiv (\forall\, G\ a\ b.\ blockDAG\ G \longrightarrow a \rightarrow^+_G b \longrightarrow (b,a) \in (A\ G))$

definition *One-Appending-Monotone*:: $(('a,'b)\ pre\text{-}digraph \Rightarrow 'a\ rel) \Rightarrow bool$
where $One\text{-}Appending\text{-}Monotone\ A \equiv$
$(\forall\, G\ G'\ a\ b\ c.\ Honest\text{-}Append\text{-}One\ G\ G'\ a \longrightarrow ((b,c) \in (A\ G) \longrightarrow (b,c) \in (A\ G')))$

definition *One-Appending-Robust*:: $(('a,'b)\ pre\text{-}digraph \Rightarrow 'a\ rel) \Rightarrow bool$
where $One\text{-}Appending\text{-}Robust\ A \equiv$
$(\forall\, G\ G'\ G''\ a\ b\ c\ d.\ Append\text{-}One\text{-}Honest\text{-}Dishonest\ G\ G'\ a\ G''\ b$
$\longrightarrow ((c,d) \in (A\ G) \longrightarrow (c,d) \in (A\ G'')))$

end
theory *Spectre-Properties*
imports *Spectre ExtendblockDAG Properties*
begin

7 SPECTRE properties

7.1 SPECTRE Order Preserving

lemma *vote-Spectre-Preserving*:
assumes $c \rightarrow^+_G b$
shows *vote-Spectre G a b c* $\in \{0,1\}$
using *assms*
proof(*induction G a b c rule*: *vote-Spectre.induct*)
case (*1 V a b c*)
then show *?case*
proof(*cases a b c V rule*:*Spectre-casesAlt*)
case *no-bD*
then show *?thesis* **by** *auto*
next
case *equal*
then show *?thesis* **by** *simp*
next
case *one*
then show *?thesis* **by** *auto*

next
case *two*
then show *?thesis*
by (*metis local.1.prems trancl-trans*)
next
case *three*
then have *a-not-gen*: ¬ *blockDAG.is-genesis-node V a*
using *blockDAG.genesis-reaches-nothing*
by *metis*
then have *bD*: *blockDAG* (*reduce-past V a*) **using** *blockDAG.reduce-past-dagbased*

three **by** *auto*
have *b-in2*: *b* ∈ *past-nodes V a* **using** *three* **by** *auto*
also have *c-in2*: *c* ∈ *past-nodes V a* **using** *three* **by** *auto*
ultimately have *c* $\rightarrow^{+}_{reduce\text{-}past\ V\ a}$ *b* **using** *DAG.reduce-past-path2 three 1*
by (*metis blockDAG.axioms*(*1*))
then have *allsorted01*:∀ *x*. *x* ∈ *set* (*sorted-list-of-set* (*past-nodes V a*)) ⟶
vote-Spectre (*reduce-past V a*) *x b c* ∈ {*0*, *1*} **using** *1 three* **by** *auto*
then have *all01*: ∀ *x*. *x* ∈ (*past-nodes V a*) ⟶
vote-Spectre (*reduce-past V a*) *x b c* ∈ {*0*, *1*}
using *subs three sorted-list-of-set*(*1*) *DAG.finite-past*
by *metis*
obtain *wit* **where** *wit-in*: *wit* ∈ *past-nodes V a*
and *wit-vote*: *vote-Spectre* (*reduce-past V a*) *wit b c* ≠ *0*
using *vote-Spectre-one-exists b-in2 c-in2 bD induce-subgraph-verts reduce-past.simps*
by *metis*
then have *wit-vote1*: *vote-Spectre* (*reduce-past V a*) *wit b c* = *1* **using** *all01*
by *blast*
obtain *the-map* **where** *the-map-in*:
the-map = (*map* (λ*i*. *vote-Spectre* (*reduce-past V a*) *i b c*)
(*sorted-list-of-set* (*past-nodes V a*)))
by *auto*
have *all01-1*: ∀ *x* ∈ *set the-map*. *x* ∈ {*0*,*1*}
unfolding *the-map-in set-map*
using *allsorted01* **by** *blast*
have ∃ *x* ∈ *set the-map*. *x* = *1*
unfolding *the-map-in set-map*
using *wit-in wit-vote1*
sorted-list-of-set(*1*) *DAG.finite-past bD subs*
by (*metis* (*no-types*, *lifting*) *image-iff three*)
then have ∃ *x* ∈ *set the-map*. *x* > *0*
using *zero-less-one* **by** *blast*
moreover have ∀ *x* ∈ *set the-map*. *x* ≥ *0* **using** *all01-1*
by (*metis empty-iff insert-iff less-int-code*(*1*) *not-le-imp-less zero-le-one*)
ultimately have *signum* (*sum-list the-map*) = *1* **using** *sumlist-one-mono* **by**
simp
then have *tie-break-int b c* (*signum* (*sum-list the-map*)) = *1* **using** *tie-break-int.simps*
by *simp*
then have *vote-Spectre V a b c* = *1* **unfolding** *the-map-in* **using** *three*

vote-Spectre.simps
by *simp*
then show *?thesis* **by** *simp*
next
case *four*
then have *all01*: $\forall$ *a2*. *a2* $\in$ *set* (*sorted-list-of-set* (*future-nodes V a*)) $\longrightarrow$ *vote-Spectre V a2 b c* $\in$ {*0*,*1*}
using *1*
by *metis*
have $\forall$ *a2*. *a2* $\in$ *set* (*sorted-list-of-set* (*future-nodes V a*)) $\longrightarrow$ *vote-Spectre V a2 b c* $\geq$ *0*
proof *safe*
fix *a2*
assume *a2* $\in$ *set* (*sorted-list-of-set* (*future-nodes V a*))
then have *vote-Spectre V a2 b c* $\in$ {*0*, *1*} **using** *all01* **by** *auto*
then show *vote-Spectre V a2 b c* $\geq$ *0*
by *fastforce*
qed
then have (*sum-list* (*map* (λ*i*. *vote-Spectre V i b c*) (*sorted-list-of-set* (*future-nodes V a*)))) $\geq$ *0* **using** *sum-list-mono sum-list-0*
by *metis*
then have *signum* (*sum-list* (*map* (λ*i*. *vote-Spectre V i b c*) (*sorted-list-of-set* (*future-nodes V a*)))) $\in$ {*0*,*1*} **unfolding** *signum.simps*
by *simp*
then show *?thesis* **using** *four* **by** *simp*
qed
qed

lemma *Spectre-Order-Preserving*:
assumes *blockDAG G*
and $b \rightarrow^+_G a$
shows *Spectre-Order G a b*
proof −
interpret *B*: *blockDAG G* **using** *assms*(*1*) **by** *auto*
have *set-ordered*: *set* (*sorted-list-of-set* (*verts G*)) = *verts G*
using *assms*(*1*) *subs fin-digraph.finite-verts sorted-list-of-set* **by** *auto*
have *a-in*: *a* $\in$ *verts G* **using** *B.reachable1-in-verts*(*2*) *assms*(*2*)
by *metis*
have *b-in*: *b* $\in$ *verts G* **using** *B.reachable1-in-verts*(*1*) *assms*(*2*)
by *metis*
obtain *the-map* **where** *the-map-in*:
the-map = (*map* (λ*i*. *vote-Spectre G i a b*) (*sorted-list-of-set* (*verts G*))) **by** *auto*
obtain *wit* **where** *wit-in*: *wit* $\in$ *verts G* **and** *wit-vote*: *vote-Spectre G wit a b* $\neq$ *0*
using *vote-Spectre-one-exists a-in b-in assms*(*1*)
by *blast*

have (*vote-Spectre G wit a b*) ∈ *set the-map*
unfolding *the-map-in set-map*
using *B.blockDAG-axioms fin-digraph.finite-verts*
subs sorted-list-of-set(*1*) *wit-in image-iff*
by *metis*
then have *exune*: ∃ *x* ∈ *set the-map*. *x* ≠ *0*
using *wit-vote* **by** *blast*
have *all01*: ∀ *x* ∈ *set the-map*. *x* ∈ {*0*,*1*}
unfolding *set-ordered the-map-in set-map* **using** *vote-Spectre-Preserving assms*(*2*) *image-iff*
by (*metis* (*no-types*, *lifting*))
then have ∃ *x* ∈ *set the-map*. *x* = *1* **using** *exune*
by *blast*
then have ∃ *x* ∈ *set the-map*. *x* > *0*
using *zero-less-one* **by** *blast*
moreover have ∀ *x* ∈ *set the-map*. *x* ≥ *0* **using** *all01*
by (*metis empty-iff insert-iff less-int-code*(*1*) *not-le-imp-less zero-le-one*)
ultimately have *signum* (*sum-list the-map*) = *1* **using** *sumlist-one-mono* **by** *simp*
then have *tie-break-int a b* (*signum* (*sum-list the-map*)) = *1* **using** *tie-break-int.simps*
by *simp*
then show *?thesis* **unfolding** *the-map-in Spectre-Order-def* **by** *simp*
qed

lemma *SPECTRE-Preserving*:
assumes *blockDAG G*
and *b* $\rightarrow^+_G$ *a*
shows (*a*,*b*) ∈ (*SPECTRE G*)
unfolding *SPECTRE-def*
using *assms wf-digraph.reachable1-in-verts subs*
Spectre-Order-Preserving
SigmaI case-prodI mem-Collect-eq **by** *fastforce*

lemma *Order-Preserving SPECTRE*
unfolding *Order-Preserving-def*
using *SPECTRE-Preserving* **by** *auto*

lemma *vote-Spectre-antisymmetric*:
shows *b* ≠ *c* ⟹ *vote-Spectre V a b c* = − (*vote-Spectre V a c b*)
proof(*induction V a b c rule*: *vote-Spectre.induct*)
case (*1 V a b c*)
show *vote-Spectre V a b c* = − *vote-Spectre V a c b*
proof(*cases a b c V rule*:*Spectre-casesAlt*)
case *no-bD*
then show *?thesis* **by** *fastforce*
next
case *equal*
then show *?thesis* **using** *1* **by** *simp*

next
case *one*
then show *?thesis* **by** *auto*
next
case *two*
then show *?thesis* **by** *fastforce*
next
case *three*
then have *ff*: *vote-Spectre V a b c* = *tie-break-int b c* (*signum* (*sum-list* (*map*
(λ*i*.
(*vote-Spectre* (*reduce-past V a*) *i b c*)) (*sorted-list-of-set* (*past-nodes V a*)))))
using *vote-Spectre.simps*
by (*metis* (*mono-tags*, *lifting*))
have *ff1*: *vote-Spectre V a c b* = *tie-break-int c b* (*signum* (*sum-list* (*map* (λ*i*.
(*vote-Spectre* (*reduce-past V a*) *i c b*)) (*sorted-list-of-set* (*past-nodes V a*)))))
using *vote-Spectre.simps three* **by** *fastforce*
then have *ff2*: *vote-Spectre V a c b* = *tie-break-int c b* (*signum* (*sum-list* (*map*
(λ*i*.
(− *vote-Spectre* (*reduce-past V a*) *i b c*)) (*sorted-list-of-set* (*past-nodes V a*)))))
using *three 1 map-eq-conv ff*
by (*smt* (*verit*, *best*))
have (*map* (λ*i*. − *vote-Spectre* (*reduce-past V a*) *i b c*) (*sorted-list-of-set*
(*past-nodes V a*)))
= (*map uminus* (*map* (λ*i*. *vote-Spectre* (*reduce-past V a*) *i b c*)
(*sorted-list-of-set* (*past-nodes V a*))))
using *map-map* **by** *auto*
then have *vote-Spectre V a c b* = − (*tie-break-int b c* (*signum* (*sum-list* (*map*
(λ*i*.
(*vote-Spectre* (*reduce-past V a*) *i b c*)) (*sorted-list-of-set* (*past-nodes V a*))))))
using *antisymmetric-sumlist 1 ff2 antisymmetric-signum antisymmetric-tie-break*
by (*metis verit-minus-simplify*(*4*))
then show *?thesis* **using** *ff*
by *presburger*
next
case *four*
then have *ff*: *vote-Spectre V a b c* = *signum* (*sum-list* (*map* (λ*i*.
(*vote-Spectre V i b c*)) (*sorted-list-of-set* (*future-nodes V a*))))
using *vote-Spectre.simps*
by (*metis* (*mono-tags*, *lifting*))
then have *ff2*: *vote-Spectre V a c b* = *signum* (*sum-list* (*map* (λ*i*.
(− *vote-Spectre V i b c*)) (*sorted-list-of-set* (*future-nodes V a*))))
using *four 1 vote-Spectre.simps map-eq-conv*
by (*smt* (*z3*))
have (*map* (λ*i*. − *vote-Spectre V i b c*) (*sorted-list-of-set* (*future-nodes V a*)))
= (*map uminus* (*map* (λ*i*. *vote-Spectre V i b c*) (*sorted-list-of-set* (*future-nodes*
V a))))
using *map-map* **by** *auto*
then have *vote-Spectre V a c b* = − (*signum* (*sum-list* (*map* (λ*i*.
(*vote-Spectre V i b c*)) (*sorted-list-of-set* (*future-nodes V a*)))))

```
    using  antisymmetric-sumlist 1 ff2 antisymmetric-signum
    by (metis verit-minus-simplify(4))
  then show ?thesis using ff
    by linarith
 qed
qed


lemma vote-Spectre-reflexive:
 assumes blockDAG V
   and a ∈ verts V
 shows ∀ b ∈ verts V. vote-Spectre V b a a = 1  using vote-Spectre.simps assms
by auto

lemma Spectre-Order-reflexive:
 assumes blockDAG V
   and a ∈ verts V
 shows Spectre-Order V a a
 unfolding Spectre-Order-def
proof −
 obtain l where l-def: l = (map (λi. vote-Spectre V i a a) (sorted-list-of-set (verts
V)))
   by auto
 have only-one: l = (map (λi.1) (sorted-list-of-set (verts V)))
   using l-def vote-Spectre-reflexive assms sorted-list-of-set(1)
   by (simp add: fin-digraph.finite-verts subs)
 have ne: l ≠ []
   using  blockDAG.no-empty-blockDAG length-map
    by (metis assms(1) length-sorted-list-of-set less-numeral-extra(3) list.size(3)
l-def)
 have sum-list l = card (verts V) using ne only-one sum-list-map-eq-sum-count
   by (simp add: sum-list-triv)
 then have sum-list l > 0 using blockDAG.no-empty-blockDAG assms(1) by simp
 then show tie-break-int a a
  (signum (sum-list (map (λi. vote-Spectre V i a a) (sorted-list-of-set (verts V)))))
= 1
   using l-def ne  tie-break-int.simps
     list.exhaust verit-comp-simplify1(1) by auto
qed


lemma Spectre-Order-antisym:
 assumes blockDAG V
   and a ∈ verts V
   and b ∈ verts V
   and a ≠ b
 shows Spectre-Order V a b = (¬ (Spectre-Order V b a))
proof −
 obtain wit where wit-in: vote-Spectre V wit a b ≠ 0 ∧ wit ∈ verts V
```

```
    using  vote-Spectre-one-exists assms
    by blast
  obtain l where l-def: l = (map (λi. vote-Spectre V i a b) (sorted-list-of-set (verts
V)))
    by auto
  have wit ∈  set (sorted-list-of-set (verts V))
    using  wit-in sorted-list-of-set(1)
      fin-digraph.finite-verts subs
    by (simp add: fin-digraph.finite-verts subs assms(1))
  then have vote-Spectre V wit a b ∈ set l unfolding l-def
    by (metis (mono-tags, lifting) image-eqI list.set-map)
  then have dm: tie-break-int a b (signum (sum-list l)) ∈ {−1,1}
    by auto
  obtain l2 where l2-def: l2 = (map (λi. vote-Spectre V i b a) (sorted-list-of-set
(verts V)))
    by auto
  have minus: l2 = map uminus l
    unfolding l-def l2-def map-map
    using  vote-Spectre-antisymmetric assms(4)
    by (metis comp-apply)
  have anti: tie-break-int a b (signum (sum-list l)) =
                − tie-break-int b a (signum (sum-list l2)) unfolding minus
      using antisymmetric-sumlist antisymmetric-tie-break antisymmetric-signum
assms(4) by metis
  then show ?thesis unfolding Spectre-Order-def using anti l-def dm l2-def
      add.inverse-inverse empty-iff equal-neg-zero insert-iff zero-neq-one
    by (metis)
qed

lemma Spectre-Order-total:
  assumes blockDAG V
    and a ∈ verts V ∧ b ∈ verts V
  shows Spectre-Order V a b ∨ Spectre-Order V b a
proof safe
  assume notB:  ¬ Spectre-Order V b a
  consider (eq) a = b| (neq) a ≠ b by auto
  then show Spectre-Order V a b
  proof (cases)
    case eq
    then show ?thesis using Spectre-Order-reflexive assms by metis
  next
    case neq
    then show ?thesis using Spectre-Order-antisym notB assms
      by blast
  qed
qed

lemma SPECTRE-total:
  assumes blockDAG G
```

shows *total-on* (*verts G*) (*SPECTRE G*)
unfolding *total-on-def SPECTRE-def*
using *Spectre-Order-total assms*
by *fastforce*

lemma *SPECTRE-reflexive*:
assumes *blockDAG G*
shows *refl-on* (*verts G*) (*SPECTRE G*)
unfolding *refl-on-def SPECTRE-def*
using *Spectre-Order-reflexive assms* **by** *fastforce*

lemma *SPECTRE-antisym*:
assumes *blockDAG G*
shows *antisym* (*SPECTRE G*)
unfolding *antisym-def SPECTRE-def*
using *Spectre-Order-antisym assms* **by** *fastforce*

lemma *Spectre-equals-vote-Spectre-honest*:
assumes *Honest-Append-One G G-A a*
and $b \in$ *verts G*
and $c \in$ *verts G*
shows *Spectre-Order G b c* $\longleftrightarrow$ *vote-Spectre G-A a b c = 1*
proof −
interpret *H*: *Append-One G G-A a* **using** *assms(1) Honest-Append-One-def* **by** *metis*
have *b-in*: $b \in$ *verts G-A* **using** *H.append-verts-in assms(2)* **by** *metis*
have *c-in*: $c \in$ *verts G-A* **using** *H.append-verts-in assms(3)* **by** *metis*
have *re-all*: $\forall v \in$ *verts G*. $a \rightarrow^+_{G\text{-}A} v$ **using** *Honest-Append-One.reaches-all assms(1)* **by** *metis*
then have *r-ab*: $a \rightarrow^+_{G\text{-}A} b$ **using** *assms(2)* **by** *simp*
have *r-ac*: $a \rightarrow^+_{G\text{-}A} c$ **using** *re-all assms(3)* **by** *simp*
consider (*b-c-eq*) $b = c$ | (*not-b-c-eq*) $\neg\ b = c$ **by** *auto*
then show *?thesis*
proof(*cases*)
case *b-c-eq*
then have *Spectre-Order G b c* **using** *Spectre-Order-reflexive Append-One-def H.Append-One-axioms assms*
by *metis*
moreover have *vote-Spectre G-A a b c = 1* **using** *vote-Spectre-reflexive*
using *H.bD-A H.app-in b-in b-c-eq* **by** *metis*
ultimately show *?thesis* **by** *simp*
next
case *not-b-c-eq*
then have *vote-Spectre G-A a b c* = (*tie-break-int b c* (*signum* (*sum-list* (*map* (*λi*.
(*vote-Spectre* (*reduce-past G-A a*) *i b c*)) (*sorted-list-of-set* (*past-nodes G-A a*))))))
using *vote-Spectre.simps Append-One.bD-A Honest-Append-One-def assms r-ab r-ac*

Append-One.app-in b-in c-in
map-eq-conv **by** *fastforce*
then have *the-eq*: *vote-Spectre G-A a b c* = (*tie-break-int b c* (*signum* (*sum-list* (*map* (λ*i*.
(*vote-Spectre G i b c*)) (*sorted-list-of-set* (*verts G*))))))
using *Honest-Append-One.reduce-append Honest-Append-One.append-past-all*

assms(*1*) **by** *metis*
show *?thesis*
unfolding *the-eq Spectre-Order-def* **by** *simp*
qed
qed

lemma *Spectre-Order-Appending-Mono*:
assumes *Honest-Append-One G G-A app*
and *a* ∈ *verts G*
and *b* ∈ *verts G*
and *c* ∈ *verts G*
and *Spectre-Order G b c*
shows *vote-Spectre G a b c* ≤ *vote-Spectre G-A a b c*
proof −
interpret *H*: *Append-One* **using** *assms*(*1*) *Honest-Append-One-def* **by** *metis*
have *app* ∈ *verts G-A* **using** *H.app-in* **oops**
end

theory *Codegen*
imports *blockDAG Spectre Ghostdag ExtendblockDAG*
begin

8 Code Generation

fun *arcAlt*:: ('*a*,'*b*) *pre-digraph* ⇒ '*b* ⇒ '*a* × '*a* ⇒ *bool*
where *arcAlt G e uv* = (*e* ∈ *arcs G* ∧ *tail G e* = *fst uv* ∧ *head G e* = *snd uv*)

fun *iterate*:: ('*a* ⇒ *bool*) ⇒ '*a set* ⇒ *bool*
where *iterate S P* = *Finite-Set.fold* (λ *r A*. *S r* ∧ *A*) *False P*

lemma (**in** *DAG*) *arcAlt-eq*:
shows *arcAlt G e uv* = *wf-digraph.arc G e uv*
unfolding *arc-def arcAlt.simps* **by** *simp*

lemma [*code*]: *blockDAG G* = (*DAG G* ∧ ((∃ *p* ∈ *verts G*. ((∀ *r* ∈ *verts G*. (*r*
→⁺$_{G}$ *p* ∨ *r* = *p*)))) ∧
(∀ *e* ∈ (*arcs G*). ∀ *u* ∈ *verts G*. ∀ *v* ∈ *verts G*.
(*u* →⁺$_{(pre\text{-}digraph.del\text{-}arc\ G\ e)}$ *v*) ⟶ ¬ *arcAlt G e* (*u*,*v*))))
using *DAG.arcAlt-eq wf-digraph-def DAG.axioms*(*1*)
digraph.axioms(*1*) *fin-digraph.axioms*(*1*) *wf-digraph.arcE blockDAG-axioms-def*

blockDAG-def
by *metis*

lemma [*code*]: *DAG G* = (*digraph G* ∧ (∀ *v* ∈ *verts G*. ¬(*v* →⁺ *G v*)))
unfolding *DAG-axioms-def DAG-def*
by (*metis digraph.axioms*(*1*) *fin-digraph.axioms*(*1*) *wf-digraph.reachable1-in-verts*(*1*))

lemma [*code*]: *digraph G* = (*fin-digraph G* ∧ *loopfree-digraph G* ∧ *nomulti-digraph G*)
unfolding *digraph-def* **by** *auto*

lemma [*code*]: *wf-digraph G* = (
(∀ *e* ∈ *arcs G*. *tail G e* ∈ *verts G*) ∧
(∀ *e* ∈ *arcs G*. *head G e* ∈ *verts G*))
using *wf-digraph-def* **by** *auto*

lemma [*code*]: *nomulti-digraph G* = (*wf-digraph G* ∧
(∀ *e1* ∈ *arcs G*. ∀ *e2* ∈ *arcs G* .
arc-to-ends G e1 = *arc-to-ends G e2* ⟶ *e1* = *e2*))
unfolding *nomulti-digraph-def nomulti-digraph-axioms-def* **by** *auto*

lemma [*code*]: *loopfree-digraph G* = (*wf-digraph G* ∧ (∀ *e* ∈ *arcs G*. *tail G e* ≠ *head G e*))
unfolding *loopfree-digraph-def loopfree-digraph-axioms-def* **by** *auto*

lemma [*code*]: *pre-digraph.del-arc G a* =
(| *verts* = *verts G*, *arcs* = *arcs G* − {*a*}, *tail* = *tail G*, *head* = *head G* |)
by (*simp add*: *pre-digraph.del-arc-def*)

lemma [*code*]: *fin-digraph G* = (*wf-digraph G* ∧ (*card* (*verts G*) > *0* ∨ *verts G* = {})
∧ ((*card* (*arcs G*) > *0* ∨ *arcs G* = {})))
using *card-ge-0-finite fin-digraph-def fin-digraph-axioms-def*
by (*metis card-gt-0-iff finite.emptyI*)

fun *vote-Spectre-Int*:: (*integer*, *integer*×*integer*) *pre-digraph* ⇒
integer ⇒ *integer* ⇒ *integer* ⇒ *integer*
where *vote-Spectre-Int V a b c* = *integer-of-int* (*vote-Spectre V a b c*)

fun *SpectreOrder-Int*:: (*integer*, *integer*×*integer*) *pre-digraph* ⇒ *integer* ⇒ *integer* ⇒ *bool*
where *SpectreOrder-Int G* = *Spectre-Order G*

fun *OrderDAG-Int*:: (*integer*, *integer*×*integer*) *pre-digraph* ⇒
integer ⇒ (*integer set* × *integer list*)
where *OrderDAG-Int V a* = (*OrderDAG V* (*nat-of-integer a*))

export-code *top-sort anticone set blockDAG pre-digraph-ext snd fst vote-Spectre-Int SpectreOrder-Int OrderDAG-Int*
in *Haskell* **module-name** *DAGS* **file** *code/*

notepad begin
let *?G* = (|*verts* = {*1*::*int*,*2*,*3*,*4*,*5*,*6*,*7*,*8*,*9*,*10*}, *arcs* = {(*2*,*1*),(*3*,*1*),(*4*,*1*), (*5*,*2*),(*6*,*3*),(*7*,*4*),(*8*,*5*),(*8*,*3*),(*9*,*6*),(*9*,*4*),(*10*,*7*),(*10*,*2*)}, *tail* = *fst*, *head* = *snd*|)
let *?a* = *2*
let *?b* = *3*
let *?c* = *4*
value *blockDAG ?G*
value *Spectre-Order ?G ?a ?b* ∧ *Spectre-Order ?G ?b ?c* ∧ ¬ *Spectre-Order ?G ?a ?c*
end

8.1 Extend Graph

declare *pre-digraph.del-vert-def* [*code*]
declare *Append-One-axioms-def* [*code*]
declare *Honest-Append-One-axioms-def*[*code*]
declare *Append-One-def* [*code*]
declare *Honest-Append-One-def* [*code*]
declare *Append-One-Honest-Dishonest-axioms-def* [*code*]
declare *Append-One-Honest-Dishonest-def* [*code*]

end
theory *Ghostdag-Properties*
imports *Ghostdag ExtendblockDAG Properties Codegen*
begin

9 GHOSTDAG properties

9.1 GHOSTDAG Order Preserving

lemma *GhostDAG-preserving*:
assumes *blockDAG G*
and $x \rightarrow^+_G y$
shows (*y*,*x*) ∈ *GHOSTDAG k G*
unfolding *GHOSTDAG.simps* **using** *assms*
proof(*induct G k arbitrary*: *x y rule*: *OrderDAG.induct*)
case (*1 G k*)
then show *?case* **proof** (*cases G rule*: *OrderDAG-casesAlt*)
case *ntB*
then show *?thesis* **using** *1* **by** *auto*

```
  next
    case one
    then have ¬ x →⁺_G y
      using subs wf-digraph.reachable1-in-verts 1
      by (metis DAG.cycle-free OrderDAG-casesAlt blockDAG.reduce-less
       blockDAG.reduce-past-dagbased blockDAG.unique-genesis less-one not-one-less-zero)

    then show ?thesis using 1 by simp
  next
    case more
    obtain pp where pp-in: pp =  (map (λi. (OrderDAG (reduce-past G i) k, i))
      (sorted-list-of-set (tips G))) using blockDAG.tips-exist by auto
    have backw: list-to-rel (snd (OrderDAG G k)) =
                    list-to-rel (snd (fold (app-if-blue-else-add-end G k)
                (top-sort G (sorted-list-of-set (anticone G (snd (choose-max-blue-set
pp)))))
                  (add-set-list-tuple (choose-max-blue-set pp))))
      using OrderDAG.simps less-irrefl-nat more pp-in
      by (metis (mono-tags, lifting))
    obtain S where s-in:
      (top-sort G (sorted-list-of-set (anticone G (snd (choose-max-blue-set pp)))))
= S by simp
   obtain t where t-in : (add-set-list-tuple (choose-max-blue-set pp)) = t by simp
    obtain ma where ma-def: ma = (snd (choose-max-blue-set pp)) by simp
     have ma-vert: ma ∈ verts G unfolding ma-def using chosen-map-simps(2)
digraph.tips-in-verts
        more(1) subs subsetD pp-in by blast
    have ma-tip: is-tip G ma unfolding ma-def
      using chosen-map-simps(2) more pp-in  tips-tips
      by (metis (no-types))
   then have no-gen: ¬ blockDAG.is-genesis-node G ma unfolding ma-def using
pp-in
        blockDAG.tips-unequal-gen more
      by metis
    then have red-bd: blockDAG (reduce-past G ma)
      using blockDAG.reduce-past-dagbased more ma-vert unfolding ma-def
      by auto
    consider (ind) x ∈ past-nodes G ma ∧ y ∈ past-nodes G ma
      |(x-in) x ∉  past-nodes G ma ∧ y ∈ past-nodes G ma
      |(y-in) x ∈  past-nodes G ma ∧ y ∉ past-nodes G ma
      |(both-nin) x ∉  past-nodes G ma ∧ y ∉ past-nodes G ma by auto
    then show ?thesis proof(cases)
      case ind
      then have x →⁺_reduce-past G ma y using DAG.reduce-past-path2 more
          1 subs
        by (metis)
      moreover have ma-tips:  ma ∈ set (sorted-list-of-set (tips G))
        using chosen-map-simps(1) pp-in more(1)
        unfolding ma-def by auto
```

```
    ultimately have (y,x) ∈ list-to-rel (snd (OrderDAG (reduce-past G ma) k))
      unfolding ma-def
      using more 1 ind less-numeral-extra(4) ma-def red-bd
      by (metis)
    then have (y,x) ∈ list-to-rel (snd (fst (choose-max-blue-set pp)))
      using chosen-map-simps(6) pp-in 1 unfolding ma-def by fastforce
  then have rel-base: (y,x) ∈ list-to-rel (snd (add-set-list-tuple(choose-max-blue-set
pp)))
      using add-set-list-tuple.simps list-to-rel-mono prod.collapse snd-conv
      by metis

    show ?thesis
      unfolding  ma-def backw s-in
      using rel-base  unfolding t-in
      using fold-app-mono-rel prod.collapse
      by metis
  next
    case x-in
    then have y ∈ set (snd (OrderDAG (reduce-past G ma) k))
    unfolding reduce-past.simps using induce-subgraph-verts Verts-in-OrderDAG

        more red-bd reduce-past.elims
      by (metis)
    then have y-in-base: y ∈ set (snd (fst (choose-max-blue-set pp)))
      unfolding ma-def using chosen-map-simps(6) more pp-in
      by fastforce
   consider (x-t) x = ma | (x-ant) x ∈ anticone G ma using DAG.verts-comp2
       subs 1  ma-tip ma-vert
       mem-Collect-eq tips-def wf-digraph.reachable1-in-verts(1) x-in
      by (metis (no-types, lifting))
    then show ?thesis proof(cases)
      case x-t
       then have (y,x) ∈  list-to-rel (snd (add-set-list-tuple (choose-max-blue-set
pp)))
        unfolding x-t ma-def
       using y-in-base add-set-list-tuple.simps list-to-rel-append prod.collapse sndI
        by metis
      then show ?thesis unfolding  ma-def backw s-in
        unfolding t-in
        using fold-app-mono-rel prod.collapse
        by metis
    next
      case x-ant
      then have x ∈ set (sorted-list-of-set (anticone G ma))
        using sorted-list-of-set(1) more subs
        by (metis DAG.anticon-finite)
      moreover have y ∈ set (snd (add-set-list-tuple (choose-max-blue-set pp)))
        using  add-set-list-tuple-mono in-mono prod.collapse y-in-base
        by (metis (mono-tags, lifting))
```

```
      ultimately show ?thesis unfolding backw
        by (metis fold-app-app-rel ma-def prod.collapse top-sort-con)
    qed
  next
    case y-in
    then have y ∈ past-nodes G ma unfolding past-nodes.simps using 1(2,3)
        wf-digraph.reachable1-in-verts(2) subs mem-Collect-eq trancl-trans
      by (metis (mono-tags, lifting))
    then show ?thesis using y-in by simp
  next
    case both-nin
   consider (x-t) x = ma | (x-ant) x ∈ anticone G ma using DAG.verts-comp2
        subs 1  ma-tip ma-vert
        mem-Collect-eq tips-def wf-digraph.reachable1-in-verts(1) both-nin
      by (metis (no-types, lifting))
    then show ?thesis proof(cases)
      case x-t
      have y ∈ past-nodes G ma using 1(3) more
          past-nodes.simps unfolding x-t
        by (simp add: subs wf-digraph.reachable1-in-verts(2))
      then show ?thesis using both-nin by simp
    next
      have y-ina: y ∈ anticone G ma
      proof(rule ccontr)
        assume ¬ y ∈ anticone G ma
        then have y = ma
          unfolding anticone.simps using subs wf-digraph.reachable1-in-verts(2)
1(2,3)
           ma-tip both-nin
          by fastforce
        then have x →⁺_G ma using 1(3) by auto
        then show False using subs  1(2)
          by (metis wf-digraph.tips-not-referenced ma-tip)
      qed
      case x-ant
     then have (y,x) ∈ list-to-rel (top-sort G (sorted-list-of-set (anticone G ma)))
       using y-ina DAG.anticon-finite subs 1(2,3) sorted-list-of-set(1) top-sort-rel
        by metis
      then show ?thesis unfolding backw ma-def  using
          fold-app-mono list-to-rel-mono2
        by (metis old.prod.exhaust)
    qed
  qed
 qed
qed


lemma ∀ k. Order-Preserving (GHOSTDAG k)
 unfolding Order-Preserving-def
```

using *GhostDAG-preserving*
by *blast*

9.2 GHOSTDAG Linear Order

lemma *GhostDAG-linear*:
assumes *blockDAG G*
shows *linear-order-on* (*verts G*) (*GHOSTDAG k G*)
unfolding *GHOSTDAG.simps*
using *list-order-linear OrderDAG-distinct OrderDAG-total assms* **by** *metis*

lemma $\forall k$. *Linear-Order* (*GHOSTDAG k*)
unfolding *Linear-Order-def*
using *GhostDAG-linear* **by** *blast*

9.3 GHOSTDAG One Appending Monotone

lemma *OrderDAG-append-one*:
assumes *Honest-Append-One G G-A a*
shows *snd* (*OrderDAG G-A k*) = *snd* (*OrderDAG G k*) @ [*a*]
proof −
have *bD-A*: *blockDAG G-A* **using** *assms Append-One.bD-A Honest-Append-One-def*
by *metis*
have *g1*: *card* (*verts G-A*) ≠ *1*
using *assms Append-One.append-greater-1 Honest-Append-One-def less-not-refl*
by *metis*
have (*tips G-A*) = {*a*} **using** *Honest-Append-One.append-is-only-tip assms* **by** *metis*
then have *tips-app*: (*sorted-list-of-set* (*tips G-A*)) = [*a*] **by** *auto*
obtain *the-map* **where** *the-map-in*:
the-map = ((*map* (λ*i*.(((*OrderDAG* (*reduce-past G-A i*) *k*)) , *i*)) (*sorted-list-of-set* (*tips G-A*))))
by *auto*
then have *m-l*: *the-map* = [((*OrderDAG* (*reduce-past G-A a*) *k*), *a*)]
unfolding *the-map-in* **using** *tips-app* **by** *auto*
then have *c-l*: *choose-max-blue-set the-map*
= ((*OrderDAG* (*reduce-past G-A a*) *k*), *a*)
by (*metis* (*no-types*, *lifting*) *choose-max-blue-avoid-empty list.discI list.set-cases set-ConsD*)
then have *bb*: *choose-max-blue-set the-map*
= ((*OrderDAG G k*), *a*) **using** *Honest-Append-One.reduce-append assms*
by *metis*
let *?M* = *choose-max-blue-set the-map*
have *anticone G-A* (*snd ?M*)= {}
unfolding *c-l*
using *assms Honest-Append-One.append-no-anticone sndI*
by *metis*
then have *eml*: (*top-sort G-A* (*sorted-list-of-set* (*anticone G-A* (*snd ?M*)))) = []
by (*metis sorted-list-of-set-empty top-sort.simps*(*1*))
then have (*fold* (*app-if-blue-else-add-end G k*)

(*top-sort G-A* (*sorted-list-of-set* (*anticone G-A* (*snd ?M*))))
(*add-set-list-tuple ?M*)) = (*add-set-list-tuple ?M*)
using *bb* **by** *simp*
moreover have *snd* (*add-set-list-tuple ?M*) = *snd* (*OrderDAG G k*) @ [*a*]
unfolding *bb*
using *add-set-list-tuple.simps Pair-inject add-set-list-tuple.elims snd-conv*
by (*metis* (*mono-tags*, *lifting*))
ultimately show *?thesis*
unfolding *the-map-in*
using *OrderDAG.simps bD-A g1 eml fold-simps*(*1*) *list.simps*(*8*) *list.simps*(*9*)
the-map-in tips-app
by (*metis* (*no-types*, *lifting*))
qed

lemma ∀ *k*. *One-Appending-Monotone* (*GHOSTDAG k*)
unfolding *One-Appending-Monotone-def GHOSTDAG.simps*
using *list-to-rel-mono OrderDAG-append-one*
by *metis*

9.4 GHOSTDAG One Appending Robust

datatype *FV* = *V1* | *V2* | *V3* | *V4* | *V5* | *V6* | *V7* | *V8* | *V9* | *V10*

fun *FV-Suc* :: *FV* ⇒ *FV set*
where
FV-Suc V1 = {*V1*,*V2*,*V3*,*V4*,*V5*,*V6*,*V7*,*V8*,*V9*,*V10*} |
FV-Suc V2 = {*V2*,*V3*,*V4*,*V5*,*V6*,*V7*,*V8*,*V9*,*V10*} |
FV-Suc V3 = {*V3*,*V4*,*V5*,*V6*,*V7*,*V8*,*V9*,*V10*} |
FV-Suc V4 = {*V4*,*V5*,*V6*,*V7*,*V8*,*V9*,*V10*} |
FV-Suc V5 = {*V5*,*V6*,*V7*,*V8*,*V9*,*V10*} |
FV-Suc V6 = {*V6*,*V7*,*V8*,*V9*,*V10*} |
FV-Suc V7 = {*V7*,*V8*,*V9*,*V10*} |
FV-Suc V8 = {*V8*,*V9*,*V10*} |
FV-Suc V9 = {*V9*,*V10*} |
FV-Suc V10 = {*V10*}
fun *less-eq-FV*:: *FV* ⇒ *FV* ⇒ *bool*
where *less-eq-FV a b* = (*b* ∈ *FV-Suc a*)

fun *less-FV* :: *FV* ⇒ *FV* ⇒ *bool*
where *less-FV a b* = (*a* ≠ *b* ∧ *less-eq-FV a b*)

lemma *FV-cases*:
fixes *x*::*FV*
obtains *x* = *V1* | *x* = *V2* | *x* = *V3* | *x* = *V4* | *x* = *V5* | *x* = *V6* | *x* = *V7* | *x* = *V8*
| *x* = *V9* | *x* = *V10*
proof(*cases x*, *auto*) **qed**

instantiation *FV* ::*linorder*

begin
definition *less-eq* ≡ *less-eq-FV*
definition *less* ≡ *less-FV*

instance
proof(*standard*)
fix *x y z* ::*FV*
show *x* ≤ *x* **unfolding** *less-eq-FV-def less-eq-FV.simps*
proof(*cases x*, *auto*) **qed**
show *x* ≤ *y* ∨ *y* ≤ *x*
unfolding *less-eq-FV-def less-eq-FV.simps*
by(*cases x rule*: *FV-cases*) (*cases y rule*: *FV-cases*, *auto*)+
show (*x* < *y*) = (*x* ≤ *y* ∧ ¬ *y* ≤ *x*)
unfolding *less-FV-def less-FV.simps less-eq-FV-def less-eq-FV.simps*
by(*cases x rule*: *FV-cases*) (*cases y rule*: *FV-cases*, *auto*)
show *x* ≤ *y* ⟹ *y* ≤ *x* ⟹ *x* = *y*
unfolding *less-FV-def less-FV.simps less-eq-FV-def less-eq-FV.simps*
by (*cases x rule*: *FV-cases*)(*cases y rule*: *FV-cases*, *auto*)+
show *x* ≤ *y* ⟹ *y* ≤ *z* ⟹ *x* ≤ *z*
unfolding *less-FV-def less-FV.simps less-eq-FV-def less-eq-FV.simps*
by (*cases x rule*: *FV-cases*)(*cases y rule*: *FV-cases*, *auto*)+
qed
end

instantiation *FV* ::*enum*
begin

definition *enum-FV* ≡ [*V1*,*V2*,*V3*,*V4*,*V5*,*V6*,*V7*,*V8*,*V9*,*V10*]

fun *enum-all-FV*:: (*FV* ⇒ *bool*) ⇒ *bool*
where *enum-all-FV P* = *Ball* {*V1*,*V2*,*V3*,*V4*,*V5*,*V6*,*V7*,*V8*,*V9*,*V10*} *P*

fun *enum-ex-FV*:: (*FV* ⇒ *bool*) ⇒ *bool*
where *enum-ex-FV P* = *Bex* {*V1*,*V2*,*V3*,*V4*,*V5*,*V6*,*V7*,*V8*,*V9*,*V10*} *P*

instance
apply(*standard*)
apply(*simp-all*)
unfolding *enum-FV-def UNIV-def*
proof −
show {*x*. *True*} = *set* [*V1*, *V2*, *V3*, *V4*, *V5*, *V6*, *V7*, *V8*, *V9*, *V10*]
proof *safe*
fix *x*::*FV*
show *x* ∈ *set* [*V1*, *V2*, *V3*, *V4*, *V5*, *V6*, *V7*, *V8*, *V9*, *V10*]
using *FV-cases* **by** *auto*
qed
show *distinct* [*V1*, *V2*, *V3*, *V4*, *V5*, *V6*, *V7*, *V8*, *V9*, *V10*] **by** *auto*
fix *P*
show *A*: (*P V1* ∧ *P V2* ∧ *P V3* ∧ *P V4* ∧ *P V5* ∧ *P V6* ∧ *P V7* ∧ *P V8* ∧ *P*

```
V9 ∧ P V10) = All P
    unfolding All-def
  proof(standard, auto, standard)
    fix x
    show P V1 ⟹
        P V2 ⟹ P V3 ⟹ P V4 ⟹ P V5 ⟹ P V6 ⟹ P V7 ⟹
    P V8 ⟹ P V9 ⟹ P V10 ⟹ P x = True
    proof(cases x rule: FV-cases, auto) qed
  qed
  show (P V1 ∨ P V2 ∨ P V3 ∨ P V4 ∨ P V5 ∨ P V6 ∨ P V7 ∨ P V8 ∨ P V9
∨ P V10) = Ex P
  proof(safe, auto)
    fix x::FV
    show P x ⟹
        ¬ P V1 ⟹
         ¬ P V2 ⟹ ¬ P V3 ⟹ ¬ P V4 ⟹ ¬ P V5 ⟹ ¬ P V6 ⟹ ¬ P V7
⟹ ¬ P V8 ⟹
         ¬ P V10 ⟹ P V9
    proof(cases x rule: FV-cases, auto) qed
  qed
qed

end

notepad
begin
 let ?G = (|verts = {V1, V2, V3, V4, V5, V6, V7, V8}, arcs = {(V2, V1),(V3, V2),(V4, V2),(V5, V2),
  (V6, V1),(V7, V6),(V8, V7)}, tail = fst, head = snd|)
  value blockDAG ?G
  value OrderDAG ?G 2
 let ?G2 = (|verts = {V1, V2, V3, V4, V5, V6, V7, V8, V9}, arcs = {(V2, V1),(V3, V2),(V4, V2),(V5, V2),
  (V6, V1),(V7, V6),(V8, V7),(V9, V3),(V9, V4),(V9, V5),(V9, V8)}, tail = fst, head
= snd|)
  value blockDAG ?G2
  value OrderDAG ?G2 2
  value Append-One ?G ?G2 V9
  value Honest-Append-One ?G ?G2 V9
 let ?G3 = (|verts = {V1, V2, V3, V4, V5, V6, V7, V8, V9, V10}, arcs = {(V2, V1),(V3, V2),(V4, V2),(V5, V2),
 (V6, V1),(V7, V6),(V8, V7),(V9, V3),(V9, V4),(V9, V5),(V9, V8),(V10, V3),(V10, V4),(V10, V5)},
   tail = fst, head = snd|)
  value Append-One ?G2 ?G3 V10
  value Append-One-Honest-Dishonest ?G ?G2 V9 ?G3 V10
  value OrderDAG ?G3 2
  value (V6, V2) ∈ GHOSTDAG 2 ?G
  value (V6, V2) ∉ GHOSTDAG 2 ?G3
end

end
```